Planning and Knowledge Representation for Softbots

by

Keith Golden

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by ____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree ____

Date ____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature____________________________

Date____________________________

University of Washington

Abstract

Planning and Knowledge Representation for Softbots

by Keith Golden

Chairperson of Supervisory Committee: *Professor Dan Weld*
Computer Science and Engineering

This thesis describes the design of a planner and knowledge representation languages for building software agents, known as softbots. While the focus of this thesis is on softbots, the ideas and algorithms presented here are general-purpose and could be applied to robotic domains as well. The major contributions are:

- The LCW (Local Closed World) knowledge representation, used to capture an agent's incomplete information about the world, which can include localized closure information, such as knowledge of all files in a directory. We present LCW inference and update procedures that are sound, fast and effective.

- The SADL action language, used to describe actions and goals available to the agent, including sensing actions and goals of acquiring new information. We define the semantics for SADL and we illustrate the expressiveness of the languages by showing the encodings of 50 UNIX commands.

- PUCCINI, a program for generating and executing plans in the presence of incomplete information. PUCCINI exploits LCW knowledge to solve universally quantified goals even in the presence of incomplete information, and uses SADL to represent a rich variety of sensors and effectors. We prove PUCCINI sound, and demonstrate its effectiveness by solving 10 representative planning problems from the UNIX domain.

TABLE OF CONTENTS

List of Figures viii

List of Tables x

Glossary xiii

Chapter 1: Introduction 1

1.1 Planning for Softbots 1

1.1.1 Planning 3

1.1.2 Knowledge Representation 4

1.2 The SADL action language 5

1.2.1 Annotations in SADL and UWL 6

1.2.2 Information Goals 7

1.2.3 Knowledge Preconditions 9

1.3 Local Closed World Knowledge 11

1.3.1 Sensor Abuse 12

1.3.2 Universally Quantified Goals 13

1.3.3 Local Closed World Knowledge 13

1.3.4 Local Closed World Reasoning 15

1.4 Reader's Guide 16

1.4.1 Typographic Conventions 16

1.4.2 UNIX commands 16

1.4.3 Road map 19

Chapter 2: Incomplete Information 20

2.1 Semantics . . . 20
2.2 Local Closed World Information . . . 22
2.3 Representing Closed World Information . . . 23
2.4 Local Closed World Inference . . . 25
2.5 Inference Method . . . 27
2.6 Summary . . . 30

Chapter 3: Action and Change 31

3.1 The Situation Calculus . . . 33
3.2 Goals . . . 34
3.2.1 Satisfaction and Maintenance Goals . . . 35
3.2.2 Knowledge Goals are Inherently Temporal . . . 36
3.2.3 Universally Quantified Goals . . . 38
3.2.4 LCW Goals . . . 40
3.2.5 Variables, Types, and Predicates . . . 40
3.2.6 Knowledge Preconditions Considered Harmful . . . 41
3.3 Effects . . . 44
3.3.1 Observational Effects . . . 44
3.3.2 Conditional Effects . . . 47
3.3.3 Uncertain Effects . . . 47
3.4 Temporal Projection . . . 50
3.4.1 Projection & the Frame Problem . . . 50
3.4.2 Regression . . . 56
3.4.3 Executability . . . 59
3.5 LCW Updates . . . 59
3.5.1 Representation of Change . . . 60

3.5.2 Information Gain 65
3.5.3 Information Loss 68
3.5.4 Changes in Domain 69
3.5.5 Initial Closed World Knowledge 71
3.5.6 Example 71
3.5.7 Computational Complexity of Updates 75
3.5.8 Optimality of Atomic Update Policies 77
3.5.9 Optimal Order of Atomic Updates 78
3.6 Summary 80

Chapter 4: Plans and Planning Problems **81**

4.1 Partially specified plans 81
4.1.1 Steps and Effects 81
4.1.2 Ordering constraints 82
4.1.3 Binding constraints 83
4.1.4 Causal links 83
4.1.5 Step execution 83
4.1.6 Plan refinement and repair 84
4.2 Interleaving Planning and Execution 84
4.3 The Planning Problem 85
4.4 Summary 87

Chapter 5: PUCCINI Algorithm **88**

5.1 Planning overview 88
5.1.1 Searching through Plan-Space 90
5.2 Handling Open Goals 91
5.2.1 Canonical Form 91

5.2.2 The Universal Base 92
5.2.3 Atomic Goals 95
5.2.4 Matching 95
5.2.5 Universally Quantified Goals 97
5.2.6 LCW Goals 101
5.2.7 Making Assumptions (Leap before you look) 102
5.3 Resolving Threats 104
5.3.1 Threats to Forall Links 105
5.3.2 Threats to LCW 106
5.3.3 Threats to **hands-off** 109
5.4 Controlling Execution 109
5.4.1 Impact of Execution 112
5.4.2 Backtracking Over Execution 114
5.4.3 Policies for Backtracking Over Execution 117
5.5 Computational Complexity 120
5.6 Summary 122

Chapter 6: Formal Properties **123**
6.1 Causality Theorems 123
6.2 Resolving flaws 126
6.3 Soundness 127
6.4 Incompleteness 129

Chapter 7: Empirical Evaluation **132**
7.1 PUCCINI 132
7.1.1 PUCCINI Goals 132
7.1.2 Plan trace 134

7.1.3 PUCCINI Actions . . . 142
7.2 LCW . . . 144
7.2.1 Factors Influencing LCW Speed . . . 146
7.2.2 Completeness . . . 146
7.2.3 Impact on Planning . . . 149
7.2.4 The Experimental Framework . . . 151
7.2.5 The Simulation Environment . . . 152
7.2.6 The Goal Distribution . . . 152

Chapter 8: Conclusions **154**
8.1 Related Work . . . 155
8.1.1 PUCCINI . . . 155
8.1.2 SADL . . . 157
8.1.3 LCW . . . 158
8.2 Future Work . . . 160
8.2.1 Contingency . . . 160
8.2.2 Exogenous Events . . . 160
8.2.3 Expressiveness . . . 162
8.2.4 Background Goals . . . 162
8.3 Summary . . . 164

Bibliography **166**

Appendix A: Proofs **174**
A.1 Proofs from Chapter 2 . . . 174
A.1.1 Proof of Theorem 2.1 (NP-hardness of LCW queries, unrestricted $\mathcal{L}$) . . . 174
A.1.2 Proof of Theorem 2.2 (Instantiation) . . . 175

A.1.3 Proof of Theorem 2.3 (Composition) 175
A.1.4 Proof of Theorem 2.4 (Conjunction) 175
A.1.5 Proof of Theorem 2.5 (Negation) 176
A.1.6 Proof of Theorem 2.6 (Disjunction) 176
A.1.7 Proof of Theorem 2.7 Incompleteness of LCW Inference Rules . 176
A.1.8 Proof of Theorem 2.8 Soundness of `QueryLCW` 176
A.1.9 Proof of Theorem 2.9 Complexity of `QueryLCW` 177
A.2 Proofs from Chapter 3 177
A.2.1 Proof of Theorem 3.1 (Successor State Axiom) 178
A.2.2 Proof of Lemma A.1 (Knowledge effects) 180
A.2.3 Proof of Theorem 3.2 (Successor state axiom for K) 180
A.2.4 Proof of Theorem 3.5 (No Correlations) 182
A.2.5 Proof of Theorem 3.6 (Successor state axiom for KNOW) . . . 183
A.2.6 Proof of Theorem 3.7 (Soundness of regression) 190
A.2.7 Proof of Theorem 3.8 (Completeness of regression) 192
A.2.8 Proof of Theorem 3.9 (Updates generated by **cause**(φ, `T`) . . . 195
A.2.9 Proof of Theorem 3.10 (Updates generated by **cause**(φ, `F`) . . 197
A.2.10 Proof of Theorem 3.11 (Updates generated by **cause**(φ, `U`) . . 197
A.2.11 Proof of Theorem 3.12 (Updates generated by **observe**(φ, tv) 198
A.2.12 Proof of Theorem 3.13 (Information Gain Rule) 200
A.2.13 Proof of Theorem 3.14 (Counting Rule) 200
A.2.14 Proof of Theorem 3.15 (Information Loss Rule) 200
A.2.15 Proof of Theorem 3.16 (Domain Growth Rule) 201
A.2.16 Proof of Theorem 3.17 (Domain Contraction Rule) 202
A.2.17 Proof of Theorem 3.18 (Tractability of Updates) 202
A.2.18 Proof of Theorem 3.19 (Minimal Information Loss) 203

A.2.19 Proof of Theorem 3.20 (Minimal Domain Growth) 203
A.3 Proofs from Chapter 6 204
A.3.1 Proof of Corollary 6.2 (Causality theorem for **satisfy**) 204
A.3.2 Proof of Lemma 6.3 205
A.3.3 Proof of Lemma 6.4 205
A.3.4 Proof of Lemma 6.5 206
A.3.5 Proof of Corollary 6.6 (Causality theorem for **initially**) 206
A.3.6 Proof of Corollary 6.7 (Causality theorem for **hands-off**) . . . 207
A.3.7 Proof of Theorem 6.8 (Correctness of HandleGoals) 209
A.3.8 Proof of Theorem 6.9 (Correctness of HandleThreats) 213
A.3.9 Proof of Lemma 6.11 (PUCCINI loop invariant (initial)) 215
A.3.10 Proof of Lemma 6.12 (PUCCINI loop invariant) 215
A.3.11 Proof of Lemma 6.12 (PUCCINI loop invariant true on termination) 216
A.3.12 Proof of Theorem 6.14 (Soundness of PUCCINI) 216

Appendix B: Softbot Domain **217**
B.1 File Operators 217
B.2 Person Operators 224
B.3 Machine Operators 235
B.4 Printer Operators 240
B.5 Web Operators 243
B.6 Display Operators 246

LIST OF FIGURES

1.1 The Internet Softbot 2

1.2 **UNIX action schema.** The SADL `ls` action (UNIX `ls -a`) to list all files in a directory. 9

2.1 `Query`, a fast algorithm for determining the agent's belief in a ground conjunction. 24

2.2 The `QueryLCW` algorithm determines whether a conjunctive LCW statement follows from the agent's beliefs as encoded in terms of the $\mathcal{M}$ and $\mathcal{L}$ databases. 29

3.1 ORIG$_n$, the set of states indistinguishable from s_0, based on the agent's knowledge in state s_n 36

3.2 EBNF specification of SADL. 49

5.1 PUCCINI Algorithm 90

5.2 Procedure HandleGoal 93

5.3 Procedure Reduce 94

5.4 Procedure Addlink 96

5.5 Most General Unifier 96

5.6 Procedure HandleThreats 108

5.7 Function IsExecutable 109

5.8 Procedure HandleExecution 110

7.1 CPU Time for LCW queries 147

7.2 Planner performance gain due to LCW 150

LIST OF TABLES

3.1 A summary of the mutually exclusive and exhaustive atomic update rules for the LCW database $\mathcal{L}$. 71

7.1 Planner statistics for ten sample goals 135

7.2 The number of executions performed by the planner with and without LCW 151

ACKNOWLEDGMENTS

I would like to thank Dan Weld and Oren Etzioni, both of whom taught me so much, including how to do research and how to communicate technical results effectively. None of the work in this thesis would have been possible without their guidance and advice. Having both of them pulling me, not always in the same direction, was tremendously valuable to me. I am in debt to Paul Beame for help with the theoretical aspects of this thesis. The proofs and formalisms would have been much less clear if not for his excellent advice. Any lack of clarity remaining is, of course, completely of my own devising. Thanks also to Omid Madani for helping with the theory.

This thesis would have been quite impossible if not for the hard work of the "Softbot hackers," a hardy band of students who were often seen bleary-eyed the mornings of demos. It was only by seeing them grapple with representational inadequacies in the Softbot that I was able to make some of the discoveries at the heart of this thesis.

This thesis would not be complete (or honest) without acknowledging the frequent help and support from Frankye Jones, the patron saint of graduate students, whose intercessions with the graduate school save us from bureaucratic purgatory, and who guides us along the path to our final reward: graduation.

My views about planning and AI have been influenced by Rich Segal's optimism, Nick Kushmerick's pessimism and Mike Williamson's cynicism, and

by numerous discussions with Tony Barrett, Denise Draper, Marc Friedman, Steve Hanks, Neal Lesh, Adam Carlson, and many others.

I would not be where I am now if not for the support and encouragement of my family – particularly my mom, who has been supportive of all my endeavors, has always been willing to listen to my crazy ideas, and always believed I could do anything I set my mind to, even getting a PhD.

I am especially grateful to Ellen Spertus, who, despite having her own thesis to finish, provided help and comfort to me when I was in the midst of frantic thesis hacking, and who gave me a motive to finish.

GLOSSARY

Following is a list of non-standard terms and symbols used in this thesis. Items are listed according to the following lexicographic order. Non-alphabetic symbols are listed first, followed by Greek letters in alphabetic order, and then letters of the Latin alphabet, also in alphabetical order.

Glossary of terms and symbols

Term	**Page**	**Description**
$\perp$	96	Returned by MGU, indicates that the two formulas do not unify.
$\leftarrow$	24	The assignment operator
$\vdash$	28	The RHS follows from the inference procedures from the LHS
$;$	82	The successor relation for actions.
$\prec$	82	The transitive closure of $;$.
α	56	An axiomatization of the agent's knowledge in the initial state.
Γ	86	A goal given to the agent
$\gamma^{*}_{\varphi}(a)$	51	Secondary precondition of effect ϕ of action a. The * stands for the type of effect. ...
$\Delta(x, \mathtt{tv1} \rightarrow \mathtt{tv2})$	60	Indicates an update to the truth value of x from $tv1$ to $tv2$. Possible truth values are `T`, `F` and `U`.
$S_p \xrightarrow{q} S_c$	83	A causal link, recording the commitment to support precondition q of action S_c with an effect of action S_p.
ε_a	44	

Glossary of terms and symbols (continued)

Term	Page	Defn
θ		Always used as a variable to represent variable binding constraints or instantiations of variables.
$\kappa_P^{tv}(a)$	51	Secondary precondition for observational effect of action a.
π_a	44	
Π_φ^a	52	The conditions under which action a preserves φ being true.
Σ_φ^a	52	The conditions under which action a makes φ true.
Υ	92	The universal base
ϕ		Always used to represent a literal.
Φ		Always used to represent a formula.
ψ		Always used to represent a literal.
Ψ		Always used to represent a formula.
$\{a\}_1^n$	33	A sequence of actions from a_1 to a_n.
$\mathcal{A}$	81	Steps (partially instantiated action schemas) in the current plan
S_c	83	The step consuming a given condition (having that condition
$\text{ACHV}(\varphi, s_0, \{a\}_1^n)$	34	Used to describe SADL goals in

Glossary of terms and symbols (continued)

Term	Page	Defn
ADL	31	Pednault's Action Description Language. as a primary or secondary precondition).
alit	91	An annotated literal in the SADL language.
S_p	83	The step producing a given condition (having that condition as an effect).
S_t	83	A step that threatens a causal link.
$\mathcal{B}inds$	81	A variable of the planning algorithm that represents variable binding constraints (*e.g.* $x = y$ or $w = \mathtt{c}$).
causal link	83	A structure recording commitment to support a condition in a given way.
cause	44	An effect annotation indicating that the literal in question will be affected by the planner.
`cd`	16	UNIX command: change the current directory.
`chmod`	16	UNIX command: change file permissions
conservative	25	A representation is conservative if it is ...
contingency planning	85	Planning with branches to deal with contingencies.
`cp`	16	UNIX command: copy a file
DO	33	A function in the situation calculus that maps a situation and

Glossary of terms and symbols (continued)

Term	Page	Defn
$\mathcal{D}$	86	A domain theory
domain theory	30	The set of actions and other domain knowledge used by the agent. an action to a new situation that results from executing the given action in the given situation.
EFF(φ, a, s)	34	Used to describe SADL effects in the situation calculus.
$\mathcal{E}$	81	The set of steps that are eligible for execution.
`F`	24	The "false" truth value
find-out	36	An annotation used by UWL, but not by SADL
`finger`	16	UNIX command: display information about user(s).
`ftp`	16	UNIX command: transfer files over the Internet
$\mathcal{G}$	89	The top-level goal given to the planner. the situation calculus.
`grep`	16	UNIX command: search for regular expression in file(s).
hands-off	35	A goal annotation indicating a maintenence goal of leaving the literal in question unchanged.

Glossary of terms and symbols (continued)

Term	Page	Defn
$\mathcal{I}$	86	The agent's knowledge about the initial state of the world
initially	37	A goal annotation specifying that the goal in question is only achieved if it is known to have held when it was given.
INSPEC	16	Internet resource: bibliographic database
IPE	85	Interleaving planning with execution.
K	21	The Kripke accessibility relation.
KNOW	21	A modal knowledge operator.
$\mathcal{L}$	23	A database representing the agent's closed-world knowledge (LCW).
LCW	22	Local Closed-World knowledge.
$\mathcal{C}$	81	Causal links in the current plan
`lpr`	16	UNIX command: print file
`lpq`	16	UNIX command: examine print queue for printer
`ls`	16	UNIX command: list files
$\mathcal{M}$	23	A database representing all positive literals that the agent knows to be true in the world.

Glossary of terms and symbols (continued)

Term	Page	Defn
Markov domain	41	A planning domain in which the effects of actions depend only on the state of the world (excluding the agent's knowledge) at the time of execution.
MGU	96	The most general unifier, for matching effects to goals.
`MREL`(φ)	70	A minimal set of LCW formulas relevant to certain updates to φ.
`mv`	16	UNIX command: move (rename) a file
`netfind`	16	Internet resource: find email address of user on Internet.
observe	44	An effect annotation indicating that the literal in question won't be affected by the planner, but it will be sensed.
open goal	90	A goal that has not yet been achieved.
$\mathcal{O}$	81	The set of ordering relations in the current plan
ORIG_n	36	The agent's knowledge (in situation s_n) about the initial state.
`PREL`(φ)	68	The set of LCW formulas potentially relevant to an update to φ.
`popd`	16	UNIX command: pop directory from stack and make it current.

Glossary of terms and symbols (continued)

Term	Page	Defn
`pushd`	16	UNIX command: change current directory, pushing old one on directory stack
`pwd`	16	UNIX command: print working (current) directory
PUCCINI	88	A program for Planning with Universal quantification, Conditional effects and Causal links, in the presence of INcomplete Information
R_a	56	The regression operator for action a.
`REL`(φ)	68	The set of LCW formulas relevant to an update to φ.
`rm`	16	UNIX command: delete file(s).
runtime	44	Said of variables, indicates that the value of the variable won't be known until execution.
$\mathcal{S}$	21	
s	20	A situation in the situation calculus.
s_0	33	The initial situation or state.
s_n	33	The situation resulting from the execution of $\{a\}_1^n$, *i.e.*, DO($\{a\}_1^n$, s_0)
s_*	86	The state of the world, not including the agent's knowledge
SADL	31	Sensory Action Description Language

Glossary of terms and symbols (continued)

Term	Page	Defn
satisficing	39	A search criterion concerned only with goal satisfaction, and not with solution quality.
secondary precondition	47	The precondition of a conditional effect.
satisfy	35	A goal annotation specifying that the goal in question can be achieved by any means. Most planners support this type of goal.
successor state axiom	52	An axiom that states the truth value of a proposition after execution in terms of the conditions holding prior to execution and the action executed.
threat	105	A step threatens a causal link when it possibly makes false the condition that the link protects.
`T`	24	The "true" truth value
`U`	24	The "unknown" truth value
UCPOP	88	A classical planner that uses a subset of Pednault's ADL action language. predecessor of PUCCINI.
Unk_{φ}	52	A unique predicate with unknown truth value, used to describe the semantics of effects with the `U` truth value.
UWL	31	University of Washington Language
`wc`	16	UNIX command: word count

Glossary of terms and symbols (continued)

Term	**Page**	**Defn**
$\mathcal{W}$	21	A database representing all positive literals that are true in the world. $\mathcal{W}$ is generally not accessible to an agent.
when	47	A keyword introducing the precondition of a conditional effect.

Chapter 1

INTRODUCTION

In the jigsaw-puzzle approach to building intelligent agents, the agent is divided into a number of parts, such as a "knowledge base" (to capture the agent's beliefs), a "planner" (to figure out what to do next), "sensors" (such as vision algorithms), and so on. Researchers are then supposed to go off into their separate corners to build their respective pieces in the hope that these pieces can be assembled into a complete agent. Most of the work in planning has been in this spirit, treating a planner not as an integral part of an agent that must find its way in a complex world, but as a function that maps a goal description into a sequence of actions, without interacting with the world in any way.

While divide-and-conquer is an effective method of tackling hard problems, as Rodney Brooks [4] complained, in paying too much attention to the piece, and not enough to the puzzle, we are in danger of solving an irrelevant problem — designing a piece that will never fit together with the other pieces to form a competent agent.

1.1 Planning for Softbots

This thesis reports on the design of a planner that was integrated from day one with a working software agent, the Internet Softbot [26], rather than consisting of an isolated puzzle-piece. Because of this design constraint, much of the work is concerned not with the planning algorithm *per se*, but with the knowledge representations needed by the agent as a whole to behave competently in its environment. Figure 1.1 shows the

architecture of the Softbot. The three main contributions of this thesis are the three shaded regions of the figure: The planner, called PUCCINI, and the representations of knowledge and actions.[1]

The word *softbot* stands for *soft*ware ro*bot*. For our purposes, a softbot is an intelligent, autonomous software agent, loosely analogous to a robot, except whereas a robot lives in the physical world, a softbot lives in a software world, such as UNIX or the Internet. So instead of having grippers, a softbot has commands like `mv`,[2] to move files around. And instead of having cameras or range finders, a softbot has sensors like `ls` to list files, or `finger` to find information about a user.

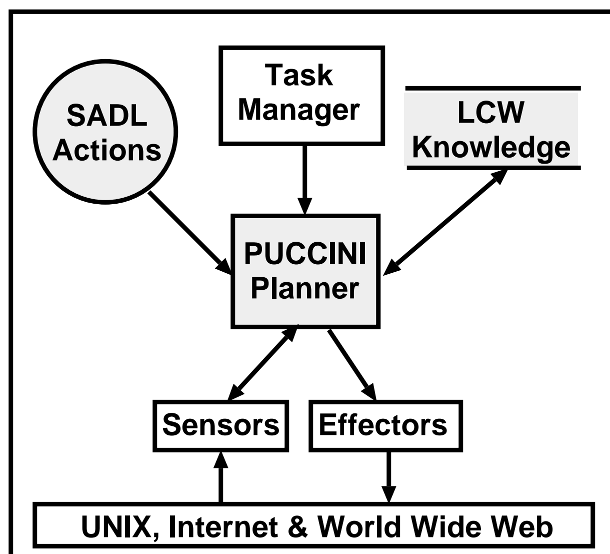

Figure 1.1: The Internet Softbot

We want our softbots to perform useful tasks for us. Ideally, we just say what we want, and rely on the softbot to figure out how to accomplish it. For example, I might ask my softbot to find a colleague's phone number or to fetch and print

[1] Portions of this thesis were previously published in [34, 25].

[2] For readers unfamiliar with UNIX, a brief description of UNIX commands can be found in Section 1.4.2.

some papers from some online repository. Goals like these require the softbot to integrate multiple resources. There is no single command that will find and print papers from an online repository; the softbot will need to first find out where the repository is, which will require sensing, then fetch the papers using `ftp`, and finally print them out. We also want our softbots to be easily adaptable to changing software environments. Internet resources come and go all the time; it should be as simple as possible to "instruct" our agents about new resources when they become available, and the softbot should continue to function when resources go away. This argues for a declarative representation; if the representation were procedural, then any time a resource changed or vanished, many procedures would potentially need to be changed, but with a declarative representation, only a single description of the resource would need to be added or removed.

1.1.1 Planning

The problem just described strongly resembles the AI planning problem. A planner takes as input a goal, a declarative description of the actions it can execute, and a description of the initial state of the world, and produces as output a sequence of those actions that, when executed, will solve the goal. It would seem like all we need to do is encode the commands and resources at the softbot's disposal as planner actions, and we can then use a planner to build our softbots. However, it's not quite that simple; most of the past work in planning has focused on toy domains, such as the Blocks World, and has made all sorts of unrealistic assumptions, such as the infamous Closed World Assumption (*i.e.*, that the agent is omniscient) and the assumption that executing actions will always produce the intended result (*i.e.*, that the agent is infallible). The challenge then, and the focus of this thesis, is to relax some of these assumptions, and extend planning technology to handle the task of building softbots.

Note that planners make all sorts of assumptions; this thesis does not attempt to

tackle all of them — only the ones that get in the way of building softbots, such as the assumptions of omniscience and infallibility just mentioned. While we don't assume the agent's information is complete, we do make the strong assumption that it is correct; for this assumption to be valid, the agent would need be be informed about any changes to the outside world that would invalidate its beliefs. Nor do we deal with *uncertainty*. Information is said to be *uncertain* when it is believed to be true with some probability. This is important, for example, when relying on noisy sensors. It is often possible in software environments to provide sensors that are noise-free, so we only address the problem of incomplete information, meaning that there are facts that the agent doesn't know, but those facts it does know, it is certain about.

1.1.2 Knowledge Representation

It turns out that when we try to scale planners up to build softbots, we run into all sorts of representational problems; the traditional representations of used by planners are inadequate for describing the actions and knowledge used by software agents. It is well-known in AI that there's a tradeoff between tractability and expressiveness. Because of this tradeoff, we would like our languages to be as expressive as we need, but no more so.

Information	⟵ Tractability	Expressiveness ⟶	
Complete	STRIPS	ADL	Situation Calculus
Incomplete	UWL	**SADL**	Moore *et al*

If we look at action languages for planning with complete information about the world (*i.e.*, making the Closed World Assumption) we find that Pednault's ADL [69] nicely captured the middle ground between inexpressive STRIPS [29] and the expressive situation calculus. [60] But before the work described here, there was no middle-ground language for planning with *incomplete* information. At one end of the spectrum, we find languages like UWL [27], which is based on STRIPS, and at the

other end, we find very expressive languages by Moore [63], Morgenstern [64, 65] and others [17, 9, 80], for which no practical planners have ever been written. In this thesis, we present SADL, a language that provides enough expressiveness to encode rich software domains, while still lending itself to practical planning.

Information	⟵ Tractability Expressiveness ⟶		
Complete	Closed World Assumption (CWA)		
Incomplete	OWA	**LCW**	Circumscription

We find a similar tractability/expressiveness spectrum for knowledge representation languages. If we look at ways of representing an agent's incomplete information about the world, we see that at one end of the spectrum, the agent can make the *Open World Assumption (OWA)*, an assumption of ignorance, which is the exact opposite of the Closed World Assumption. At the other extreme, the agent could use some form of circumscription [59], which in general is undecidable. Once again, we would a like representation in between these two extremes. We provide such a representation, in the form of *Local Closed World* knowledge (LCW).

Since we want to use a planner to build softbots, we obviously need to represent actions that the softbot will use, such as `ls` and `finger`, and goals that we would like the softbot to solve, such as "Rename `paper.tex` to `kr.tex`" or "Print all files in directory `papers`." We would like the softbot to be able to solve these goals even in the presence of incomplete information. For example, the softbot may not know what file has the name `paper.tex`, and it may not know what files are in the directory `papers`. While these goals and actions are simple, no implemented planner prior to the one reported in this thesis could express and reason about them them adequately.

1.2 The SADL *action language*

SADL is based on both UWL [27] and ADL [69], both of which are based on STRIPS. UWL extends strips by representing incomplete information and sensing. ADL extends

strips by adding universal quantification and conditional effects. By combining both languages, SADL can represent goals and actions that neither alone can represent. For example, by combining universal quantification (from ADL) with observational effects (from UWL), SADL can represent actions, like `ls`, that return an unbounded amount of information (listing *all* files in a directory, regardless of how many there are).

1.2.1 Annotations in SADL *and* UWL

Both SADL and UWL extend STRIPS by adding annotations to goals and effects. The following are the annotations common to both languages. Goal annotations include **satisfy** and **hands-off**. **satisfy** means "achieve the goal by whatever means possible." Readers familiar with planning will note that all goals in a classical planner are, in fact, **satisfy** goals. **hands-off** indicates a maintenance goal. It means "don't change this condition under any circumstances."

Effect annotations are **cause** and **observe**. **cause** effects describe changes to the world, and are just like all effects in a classical planner. **observe** effects describe changes limited to the agent's knowledge about the world, and are used to encode sensors.

To see how these annotations work in practice, consider the goal of finding and deleting some file named `core`. There are two components to this goal; we want to find a file named `core`, and we want that file to be deleted. We might be tempted to represent this goal as: **satisfy**(name(f, `core`)) $\wedge$ **satisfy**(deleted(f)). But if we do, we'll have a problem. Since **satisfy** means achieve the goal by any means, one way of achieving **satisfy**(name(f, `core`)) would be to rename some existing file, such as `thesis.tex` to `core`: `mv thesis.tex core`. But that would be a disaster, since then `thesis.tex` would be deleted, which was not the intent of the goal. One way to prevent this would be to state explicitly that the name of the file is not allowed to change: **hands-off**(name(f)). The **hands-off** will prevent the agent from using the UNIX `mv` command to rename some existing file to `core`, but still allows the

agent to passively sense the file's name, using an effect annotated with **observe**. Note that the combination of **satisfy** and **hands-off** can be interpreted as "look, but don't touch," and is one possible interpretation of an information goal, or a goal of acquiring information about the world. In fact, this is the interpretation used by the UWL language. We argue, though, that this interpretation is overly restrictive.

1.2.2 Information Goals

The reason it is overly restrictive is that information goals are inherently temporal. Whenever I ask for the truth value of some fluent (*i.e.*, a proposition that changes over time), the question is simply ambiguous unless I specify the time at which I want to know that fluent's truth value. For any information goal, there are two time points that are potentially of interest: the time when the truth value of the fluent is sampled and the time when the information about the fluent is returned. For example, I could ask someone to tell me *now* who was president *in 1883*, or I could ask someone to tell me *tomorrow* who is president *now*. Going back to the example of finding the file named `core`, what I meant to say was "Tell me *as soon as possible* the file that's *now* named core. The reason the **satisfy** + **hands-off** ("look, but don't touch") interpretation of this goal works is that the **hands-off** forces the file to have the same name at both time points (now and when the goal is achieved) and at all times in between, so whatever time I was interested in, I will get the correct answer. However, in general, using **hands-off** is overly restrictive.

To see why, consider another goal: "Rename `paper.tex` to `kr.tex`" The interesting thing about this goal is that we're using the name of the file, `paper.tex` as a designator, to identify it to our agent, and then asking that the designator be changed. There are two different time points implicit in this goal: the time the file is named `paper.tex` and the time the file is named `kr.tex`. The UWL goal annotations **satisfy** and **hands-off** can only express goals with respect to a single time point: the time the goal is achieved by the agent. This limitation is not unique to UWL; most

other action languages have the same restriction. We *can* represent the above goal, however, by adding a minimal notion of time to the language. We do this with the **initially** annotation, which refers explicitly to the time when the goal was given the the agent. With **initially**, we can express the above goal as follows: **initially**(name (f, `paper.tex`)) $\wedge$ **satisfy**(name (f, `kr.tex`))

Note that **initially** gives us an absolutely minimal representation of time. We can only express goals with respect to two time points: the time the goal was given to the agent and the time when the goal was achieved. We could also express this goal with temporal logic. The reason we don't is that planning with temporal logic tends to be computationally expensive, whereas the **initially** annotation adds no computational overhead to the planner. Furthermore, it captures the kinds of goals that we're interested in.

Going back to the goal of finding and deleting the file named `core`, what we really want to say is "find the file initially named core, and delete that file":

initially(name (f, `core`)) $\wedge$ **satisfy**(deleted (f))

We don't need **hands-off**, since changing the name of the file won't help identify the file initially named `core`; it will only obscure the identity of that file. Furthermore, once the agent has identified the correct file, there's no reason *a priori* that it should avoid renaming it afterward. If renaming the file helps to solve the goal, it should go ahead and do so; it will still delete the correct file.

Using **initially** also allows us to represent *tidiness goals* [87]: goals of having the agent clean up after itself. For example, I keep my postscript files compressed, to save disk space. If my agent prints a file for me, I want to make sure it recompresses the file afterward. I can achieve this by assigning the variable tv to the truth value of the proposition that the file is initially compressed, and requiring that the proposition have the same truth value when the goal is achieved:

initially(compressed ($paper$), tv) $\wedge$

```
action ls(d)
    precond:  satisfy(current.shell(csh)) ∧
              satisfy(protection(d, readable)) ∧
    effect:   ∀ !f when in.dir(!f, d)
                ∃ !p, !n
                  observe(in.dir(!f, d)) ∧
                  observe(pathname(!f, !p)) ∧
                  observe(name(!f, !n))
```

Figure 1.2: **UNIX action schema.** The SADL `ls` action (UNIX `ls -a`) to list all files in a directory.

satisfy(printed (*paper*)) ∧

satisfy(compressed (*paper*), *tv*)

However I don't need to specify the truth value of the proposition any time in between. This allows the agent to uncompress the file, print it, then recompress it.

By combining universal quantification with observational effects, we can also represent actions that return an unbounded amount of information about the world. Figure 1.2 shows the SADL encoding of the UNIX command `ls -a`, which lists *all* of the files in some directory d. The key thing to note is that there is a universally quantified observational effect. For *all* files f in directory d, the agent will observe the file's name, length, etc.

1.2.3 Knowledge Preconditions

One of the representational problems we grappled with when designing the Softbot concerned the use of *knowledge preconditions* to encode the information needed by the agent to execute an action. In the process, we uncovered some subtle problems

underlying the use of knowledge preconditions.

Moore [63] introduced two types of knowledge preconditions needed by an agent trying to achieve some condition P by executing an action. Namely,

1. The agent needs to know what that action is (*i.e.*, it needs an unambiguous executable description of the action) and

2. The agent needs to know that executing the action will in fact achieve P.

The first case is not a problem in our representation because the agent always has an unambiguous description of the actions it executes. For an example of the second case, suppose we want to call Avogadro by executing the action dial.phone (602-1023). In order to know that this action to has the intended result, we need to know that 602-1023 is indeed Avogadro's number and, furthermore, that he'll be home. Note that this presupposes that the agent must know in advance that the action will have its intended outcome. This is a perfectly reasonable requirement for classical planners, but is completely unreasonable when planning with incomplete information.

To see what the problem is, we will first consider Moore's example of opening a safe, when we happen to have the combination to the safe written on a piece of paper. Our goal is for the safe to be opened, which we can achieve by dialing the safe's combination: dial(n). But in order to know that the action will succeed, we need to know that n is the combination of the safe. We can achieve this goal by reading the piece of paper. Thus, we may be tempted to specify knowing the combination as a precondition of the dial action.

But let's consider another example: suppose we want to find out whether the combination of the safe is 31-24-15. We can obviously use the dial action to achieve this goal; namely, dial 31-24-15. If the safe opens, then that was the right combination. However, if we specify knowing the safe's combination as a precondition of

dial, there's no way we can use dial to achieve our goal, because dial will have the *precondition* of knowing that the combination is 31-24-15, which is what we were trying to find out in the first place.

Now consider an even more extreme example: suppose we want to open the safe, but we don't know the combination and don't have it written down anywhere. The goal is *still* solvable: There is only a finite number of combinations; why not try all of them? For point of reference, Richard Feynman estimated that he could open a safe using this method in about four hours on average. [28]

While rigid knowledge preconditions are clearly a problem, it is still necessary for an agent to gather information: to reduce search or avoid dangerous mistakes. Most importantly, the agent needs sufficient knowledge so that after it has completely executed its plan, it knows whether it has achieved its goal. However, it is not always necessary, before executing any given action, to know that the action will have its intended result. In the examples above, the agent doesn't need to know, before executing the dial action, that the action will succeed. It suffices to verify afterward that the action succeeded. As we discuss in Section 3.2.6, the solution is to eliminate rigid knowledge preconditions, and provide the planner with a flexible means (discussed in Section 5.2.7) for adopting *subgoals* of obtaining information when needed and for making assumptions that will be verified later.

1.3 Local Closed World Knowledge

Before we discuss Local Closed World knowledge (LCW), we will first motivate it by discussing the obvious alternatives. Classical planners make the Closed World Assumption (CWA). That is, their representation consists entirely of a list of facts that are true, and anything not on that list is assumed to be false. This is equivalent to assuming that they know everything. This assumption is obviously unrealistic for real-world agents.

The solution adopted in many planners that deal with incomplete information (*e.g.*, [1, 67, 27, 47, 73]) is to make the opposite assumption, which we call the Open World Assumption (OWA). That is, the agent has a list of facts that are true and facts that are false, and anything not listed is assumed to be unknown. While the OWA is obviously an improvement over the CWA, there are some problems with it as well: Planners that make the OWA fall victim to a problem called *Sensor Abuse*, and they can't solve universally quantified goals.

1.3.1 Sensor Abuse

The term Sensor Abuse [58, 61] was first used to describe robots that just don't know when to stop sensing. The problem also exists in software domains, and is a particular problem with planner-based agents. The problem with planners is that they're so darn systematic. Often there are many different sensing actions that will find the same bit of information, and there may be many viable information-gathering plans that contain many of the same sensing actions. A persistent planner that doesn't know when to quit might consider a huge number of plans that gather the same bits of information. We would like our agents to realize that after executing the command `find / -name foo`, which finds all files on the filesystem named foo, that

- Executing `ls bin` won't reveal more files named `foo`.
- Executing `ls tex` won't reveal more files named `foo`.
- On the other hand, going to a Web search engine like Alta Vista *may* reveal more files named foo, since Alta Vista returns the contents of many filesystems, not just one.

A planner that makes the OWA cannot perform this sort of reasoning. In fact, even after executing `ls bin` and finding no files, the agent will not be able to conclude

that there is no file named `foo` in `bin`. While this may seem odd, remember that the agent's entire knowledge representation consists of a list of things that are true and a list of things that are false; and both lists are necessarily finite. However, there are infinitely many possible files that are *not* in `bin`. Unless the agent specifically listed the fact that `foo` is not in `bin`, it doesn't know that `foo` is not in `bin`.

1.3.2 Universally Quantified Goals

Another problem with planners that make the OWA is that they can't solve universally quantified goals. To see why this is, let's consider how a classical planner solves a universally quantified goal. Given the of putting all blocks on the table, assuming the only blocks are A, B and C, then the planner will replace the original goal with the goal of putting A on the table, putting B on the table and putting C on the table. This approach obviously depends on the agent knowing all the blocks, but that's no problem for a classical planner, since the CWA means it knows everything.

A planner that makes the OWA can never conclude that it knows all the blocks, thus it can never solve a goal such as "put all blocks on the table."

1.3.3 Local Closed World Knowledge

What we need, obviously, is a way to represent the fact that the agent knows all blocks on a table, all files in a directory, etc. We call this knowledge *local* closed world knowledge, or LCW. Formally speaking, LCW is a restricted form of circumscription, but unlike full circumscription, it provides fast inference and updates. Furthermore, LCW is specifically tailored to the action languages used by modern planners, like the SADL language described in this thesis. The synergy between SADL and LCW is especially nice; LCW knowledge can be naturally inferred from SADL encodings of actions such as the UNIX command `ls`, which lists *all* files in a directory.

With LCW, we can represent a statement like "I know all files in directory `bin`"

quite simply. If the relation in.dir(f, `bin`) means file f is in directory `bin`, then the formula

$$\text{LCW}(\text{in.dir}(f, \texttt{bin}))$$

means the agent knows all files f satisfying that relation, *i.e.*, all files in `bin`. This is equivalent to saying that for all files f, either the agent knows that f is in `bin` or the agent knows that f is not in `bin`. This latter statement corresponds to an infinite number of false statements; it was precisely the inability to represent an infinite number of false facts that created a problem with the OWA.

We represent this information using two databases: $\mathcal{M}$ and $\mathcal{L}$. $\mathcal{M}$ is a database of ground literals, and is exactly the same as the representation used by the agents that make the OWA. That is, we list facts that are true and facts that are false. We augment this database with another database, $\mathcal{L}$, which contains LCW formulas that describe the contents of $\mathcal{M}$. For example, $\mathcal{M}$ might consist of the following facts:

- `in.dir(bar, papers)`
- `in.dir(core, papers)`
- ¬ `executable(core)`

And $\mathcal{L}$ might consist of the lone LCW formula LCW(in.dir(f, `papers`)), meaning that the agent knows all files in `papers`. Performing inference using these databases is straightforward. Suppose we ask whether some file `foo` is in directory `papers`. The agent can check in $\mathcal{M}$ to see whether there's any statement to the effect that `foo` is or is not in papers. If there were such a statement, it would just return that information. Since no such fact is stored in $\mathcal{M}$, the agent checks $\mathcal{L}$ and discovers that it knows all files in `papers`. Since it knows all the files in `papers`, and it doesn't know about file `foo`, it follows that `foo` is not in `papers`. If there hadn't been such an LCW formula, the agent would have concluded that it didn't know.

1.3.4 Local Closed World Reasoning

The notion of closed world reasoning is not new; the information in $\mathcal{L}$ LCW database is equivalent to the "closed roles" found in knowledge-representation systems such as CLASSIC [2] and LOOM [3], to predicate completion axioms [8, 46] and, as we mentioned, circumscription axioms [59, 56]. However, none of the earlier work was well-suited to the needs of a planner-based agent, *i.e.*, the ability to express and efficiently reason about the changes to the agent's closed-world knowledge that result from executing actions. This reasoning requires the agent to be able to quickly answer questions like the following:

Inference:

- If I know all files in `tex`, and I know the size of every file, then do I know the size of every file in `tex`?

Updates:

- If I know the size of every file in `tex`, and I remove a file from `tex`, do I still know the size of every file in `tex`?

- What if I *add* a file to `tex`?

In Chapters 2 and 3, we discuss how this reasoning is performed. Obtaining fast LCW reasoning is a challenge, since (as we mentioned) full circumscription is undecidable, and (as we will discuss) even for propositional theories, LCW reasoning, over arbitrary formulas containing disjunction and negation, is NP-hard. Since we need fast LCW reasoning, we limit LCW formulas to positive first-order conjunctions. In Chapters 2 and 3, we show that this representation gives us polynomial-time inference and updates. In Chapter 7, we show empirically that this reasoning is fast.

1.4 Reader's Guide

1.4.1 Typographic Conventions

The following conventions are used throughout this thesis. The names of languages or algorithms described in the literature or presented in this thesis are printed in SMALL CAPITALS. Symbols in *italics* denote variables and symbols in `typewriter` font denote constants. Special keywords and annotations are in **bold**. Names of procedures are in san serif.

For the sake of clarity, symbols used to denote variables representing various sorts of objects are always used consistently throughout this thesis. Φ and Ψ are always used to denote formulas and φ and ϕ are used to represent literals or atomic formulas The Greek letters θ, σ and α are used to denote substitutions, or mappings from variables to constants, and $\Phi\theta$ denotes the result of applying the substitution θ to the formula Φ. a is used to indicate an action and s is used to indicate a *situation*, or state of the world.

1.4.2 UNIX commands

The examples in this thesis all come from the UNIX and Internet domain (example operators from this domain can be found in Appendix B). For those readers who are unfamiliar with UNIX, we provide a brief explanation of the UNIX commands mentioned in examples in this thesis. Those readers familiar with UNIX will want to skip ahead to page 19. Variable arguments to commands are shown in *italics*.

- `cd` *dir* changes the *current directory* to *dir*. Many commands treat the current directory as an implicit argument.

- `chmod` *perm file* changes the read, write and execute permissions of *file*, assuming the user owns the file in question. There are three categories of users who can be given or denied permission to access a file: the "user" who owns

the file, the "group" to which the file belongs,[3] and "other," meaning everyone else. A file is said to be "group writable" if members of the group associated with the file have write permission for the file. `chmod *` uses the UNIX wildcard character `*` to change the permissions on all files in the current directory.

- `cp` *source dest* makes a copy of the file *source* in the specified file or directory, *dest*.

- `finger` *user*[`@`*host*] provides information about users, including name, email address and whether the user is currently logged on. If the user is logged on, `finger` will display how long the user has been "idle" (hasn't typed anything). If the user isn't logged on, `finger` will display how long ago the user last logged on. `finger` also displays the contents of a file, named "`.plan`", in the user's home directory. This file usually contains other personal information, such as phone number, address, and funny quotes the user would like to share.

- `ftp` *host* is used to transfer files between different computers on the Internet.

- `grep` *regexp file*`+` lists all occurrences of the regular expression *regexp* in the given file(s). The regular expression could simply be a string, in which case `grep` lists all occurrences of that string in the file(s).

- INSPEC is a bibliographic database for science and engineering publications. It is accessible indirectly by means of `telnet`, a program for connecting to another computer on the Internet.

- `lpr` *file* `-P`*printer* schedules *file* to be printed to *printer*.

[3] A group is a collection of users, such as all people working on a given project, or all graduate students in a department. Users can belong to multiple groups, but every file belongs to exactly one group. A user is free to give members of the group no privileges with respect to a file.

- `lpq -P`*printer* is used to find out what files are scheduled to print, or currently printing, on *printer*.

- `ls -la` *file|dir* lists all files in *dir* [4] or lists information about *file*.

- `mv` *source dest* changes the pathname of *source* to *dest*, which could involve changing its name, parent directory, or both.

- `netfind` is an Internet resource accessed by the `telnet` command. It provides a guess of a user's email address given that user's last name and location. Location can be specified by institution, city, state, country, etc. The exact information provided is up to the user, but if the location is either under-constrained or over-constrained, `netfind` will fail.

- `pushd` *dir* is like `cd` except that before changing the current directory to *dir*, it stores the (original) current directory on a stack.

- `popd` is like `cd`, except that instead of taking a directory argument, it pops the directory argument from the top of the stack written to by `pushd`, and makes that directory current. Thus `popd` reverses the effect of `pushd`.

- `pwd` displays the current directory.

- `rm` *file* deletes *file*. `rm *` deletes all files in the current directory.

- `wc` lists the name, line count, word count and character count of a given file. `wc *` provides the same information for all files in a directory.

---

[4] Along with sundry information about the files, such as size, owner, permissions, and time of last modification to be listed.

1.4.3 Road map

A glossary appears on page xiii, providing brief definitions of all nonstandard terms and symbols used in this thesis, along with page numbers indicating where those terms are introduced.

The remainder of this thesis is organized as follows. Chapter 2 discusses the representation of the agent's incomplete knowledge about the state of the world. Chapter 3 discusses the representation of actions. Chapter 4 describes how these actions are combined to form plans. Chapter 5 describes the XII planner, which uses these representations of knowledge and actions to create and execute plans that achieve a user's goals. Chapter 6 discusses the formal properties of the representations and algorithms, including soundness and completeness. Chapter 7 offers empirical evaluation of the algorithms presented in the previous chapters. Chapter 8 describes related and future work.

Chapter 2

INCOMPLETE INFORMATION

2.1 Semantics

We define the semantics of knowledge and action in terms of the situation calculus [60], a first-order logic used to capture changes to the world that come about by the execution of actions. We will discuss the aspects of the situation calculus that concern action and change in Chapter 3. Here we focus on the representation of an agent's incomplete knowledge about the world.

The situation calculus is really a discipline within first-order predicate calculus for representing the state of the world as a function of time. A *situation* is essentially a state of the world at a given time. By using situations as arguments to relations, we can make statements about facts that hold in particular situations (*i.e.*, at particular times), or about the relationship between two (real or hypothetical) situations. A *fluent* is a proposition whose truth value may change over time. Every fluent, $\varphi(x)$, takes an additional argument, namely a situation, s. $\varphi(x, s)$ represents the statement that $\varphi(x)$ holds in situation s. By convention, s is always the last argument of φ, so we will freely add or drop the s, depending on whether we are referring to φ in a particular situation. Thus, if $\text{in.dir}(f, d)$ means file f is in directory d, $\text{in.dir}(f, d, s)$ means this fact holds in situation s.

Following [68, 31, 80] and many others, we formalize an agent's incomplete information with a set of possible world states that are consistent with its information. At any given time there's one actual situation, s, which holds at that time. For any ground, atomic sentence φ, either $\varphi(s)$ or $\neg\varphi(s)$. Hence, the set of ground facts

holding in situation s forms a complete logical theory, which we denote $\mathcal{W}$. However, if the agent's knowledge is incomplete then it can't know for certain that the actual situation is s. There are many other situations that, as far as the agent knows, might hold instead. Following [80] and many others, we represent these situations using the predicate K. $\mathrm{K}(s', s)$ is true if and only if it is consistent with the agent's knowledge in situation s to believe that the situation could in fact be s'. In other words, $\{s' | \mathrm{K}(s', s)\}$ denotes the set of all possible worlds consistent with the agent's knowledge in situation s. We assume that an agent's knowledge is correct, so the actual situation is always considered possible by the agent $(\forall s.\mathrm{K}(s, s))$, We define $\mathrm{KNOW}(\varphi, s) \equiv \forall s'.\mathrm{K}(s', s) \Rightarrow \varphi(s')$, *i.e.*, φ is true in all worlds consistent with the agent's knowledge.

While the actual world, represented by the theory $\mathcal{W}$, is inaccessible to the agent, reasoning directly with sets of possible worlds is impractical for real-world applications, so we introduce $\mathcal{S}$ to denote the *incomplete* theory of ground literals known by the agent.

$$\mathcal{S} \equiv \{\varphi \ \mid \ \mathrm{KNOW}(\varphi, s)\}$$

We say that the agent possesses complete information when $\mathcal{S} = \mathcal{W}$. Incomplete information means that there are facts, φ, such that neither $\mathrm{KNOW}(\varphi, s)$ nor $\mathrm{KNOW}(\neg\varphi, s)$; in this case we say φ is unknown to the agent. We say that an atomic formula, φ, has truth value `T` if $\mathrm{KNOW}(\varphi, s)$, has truth value `F` if $\mathrm{KNOW}(\neg\varphi, s)$, or has truth value `U` (unknown) otherwise. Note that $\mathcal{S}$ is considerably less expressive than the possible-worlds representation, since it only contains individual ground literals. We cannot express facts like "Either it is raining and Fido is wet, or it is sunny and Fido is dry." The most we will be able to conclude from $\mathcal{S}$ is that we don't know if it's raining and we don't know if Fido is wet. However, despite these restrictions, $\mathcal{S}$ is still impractical to reason with, since the number of ground literals may be infinite.

2.2 Local Closed World Information

We say that an agent has *Local Closed World information* (LCW) relative to a logical formula Φ if every ground sentence that unifies with Φ is either known to be true or is necessarily false:

$$\text{LCW}(\Phi) \equiv (\text{KNOW}(\Phi\theta)) \vee (\text{KNOW}(\neg\Phi\theta)) \text{ for all ground substitutions } \theta \qquad (2.1)$$

In essence, this definition specifies which *parts* of the logical theory, $\mathcal{S}$, are complete (cf. [24] and others). Note that since $\mathcal{S}$ is a subset of $\mathcal{W}$, the definition of LCW amounts to a limited correspondence between the agent's knowledge about the world, represented by $\mathcal{S}$, and the facts that actually hold in the world, represented by $\mathcal{W}$. As a concrete example, given that `in.dir(`f`,`d`)` means "The parent directory of file f is directory d," then we can encode the fact that an agent knows all the files in the directory `/kr94` with:

$$\text{LCW}(\texttt{in.dir}(f, \texttt{/kr94}))$$

If the only files that the agent knows to be in `/kr94` are `paper.tex` and `proofs.tex`, then this LCW formula is equivalent to the following implication:

$$\forall f,\ \texttt{in.dir}(f,\ \texttt{/kr94}) \rightarrow$$
$$(f = \texttt{paper.tex}) \vee (f = \texttt{proofs.tex})$$

An LCW formula can also be understood in terms of circumscription [56]. For the example above, one defines the predicate `P(`x`)` to be true exactly when `in.dir(`x`, /kr94)` is true, and circumscribes `P` in the agent's theory.

While our work can be understood within the circumscriptive framework, our implemented agent requires the ability to infer and update[1] closed world information *quickly*. We have developed computationally tractable closed-world reasoning

[1] Following [41, 42] we distinguish between *updating* a database and *revising* it. We assume that

and update methods, applicable to the restricted representation languages used by modern planning algorithms. In the next section, we explain how to represent LCW knowledge in a manner that facilitates efficient inference. Section 2.4 describes the theory underlying LCW inference and Section 2.5 develops and analyzes an algorithm for LCW inference using these syntactic structures described below.

2.3 Representing Closed World Information

In this section, we explain how our agent represents its incomplete information about the world, and how it represents LCW in this context. Clearly, an agent cannot represent all possible situations (a potentially infinite set of large structures) explicitly. Nor can one represent $\mathcal{S}$ explicitly, since this theory can contain an infinite number of sentences.

Instead we represent the facts known by the agent with a partial database, $\mathcal{M}$, of ground literals. Formally, $\mathcal{M}$ is a subset of $\mathcal{S}$; if $\varphi \in \mathcal{M}$ then $\varphi \in \mathcal{S}$. Since $\mathcal{S}$ is incomplete, the Closed World Assumption (CWA) cannot be applied to $\mathcal{M}$. The agent cannot automatically infer that any atomic formula absent from $\mathcal{M}$ is false. Thus, the agent is forced to represent false facts in $\mathcal{M}$ explicitly, as sentences tagged with the truth value `F`.

This observation leads to a minor dilemma: the agent cannot explicitly represent in $\mathcal{M}$ *every* sentence it knows to be false (there is an infinite number of files *not* in the directory `/kr94`). Yet the agent cannot make the CWA. We adopt a simple solution: we represent local closed world information explicitly as a meta-level database, $\mathcal{L}$, containing localized closure axioms of the form LCW(Φ); these record *where* the agent has closed world information. Together, the $\mathcal{M}$ and $\mathcal{L}$ databases specify an agent's state of incomplete information about the world (*i.e.*, they constitute a partial rep-

our agent's knowledge is correct at any given time point, hence there is no need to revise it. When the world changes, however, the agent may need to update its theory to remain in agreement with the world.

```
function Query(Φ, M, L): 3-Boolean
1    let Result ← T
2    let LCW ← QueryLCW(Φ, M, L)
3    for each atomic conjunct φ ∈ Φ do begin
4        if ¬φ ∈ M then return F
5        else if φ ∉ M then
6            if LCW then return F
7            else let Result ← U
8    end(* for *)
9    return Result
```

Figure 2.1: `Query`, a fast algorithm for determining the agent's belief in a ground conjunction. `Query` returns the truth value of Φ, if Φ can be deduced from $\mathcal{M}$. Otherwise it returns either `F`, if `QueryLCW`(Φ) succeeds, or `U` if `QueryLCW` fails (`QueryLCW` is defined in Figure 2.2). We use the notation $\varphi \in \Phi$ to signify that φ is one of Φ's conjuncts.

resentation of $\mathcal{S}$).

When asked whether it believes an atomic sentence φ, the agent first checks to see if φ is in $\mathcal{M}$. If it is, then the agent responds with the truth value (`T` or `F`) associated with the sentence. However, if $\varphi \notin \mathcal{M}$ then φ could be either `F` or unknown (truth value `U`). To resolve this ambiguity, the agent checks whether $\mathcal{L}$ entails LCW(φ). If so, the fact is `F`; otherwise it is `U`. Figure 2.1 formalizes this intuitive procedure by providing pseudocode for the `Query` algorithm.

Note that the agent need not perform inference on $\mathcal{M}$, since it contains only ground literals, but it *may* need to perform some deduction on its LCW sentences. To make LCW inference and update tractable, we restrict the formulas in $\mathcal{L}$ to conjunctions of positive literals. As a result, we lose the ability to represent LCW statements that contain negation or disjunction such as "I know the protection of all files in `/kr94` *except* the files with a `.dvi` extension." Thus, for any consistent $\mathcal{M}$, $\mathcal{L}$ pair,

there exists an $\mathcal{S}$ that entails the same set of LCW sentences, but the converse is false. On the other hand, for any theory $\mathcal{S}$, there exists (at least one) pair of databases $\mathcal{M}$, $\mathcal{L}$ that represents a strict subset of the sentences in $\mathcal{S}$. We call such a $\mathcal{M}$, $\mathcal{L}$ pair a *conservative representation* of $\mathcal{S}$.

Restricting the expressiveness of $\mathcal{L}$ provides significant efficiency gains. To see this, consider a singleton LCW query such as LCW($\texttt{in.dir}(f, \texttt{/kr94})$). If $\mathcal{L}$ contains only positive conjunctions, the query can be answered by examining only singleton LCW assertions indexed under the predicate `in.dir`. If negation is allowed, however, then a combination of multiple LCW sentences has to be explored. For instance, $\text{LCW}(\Phi \wedge \Psi) \wedge \text{LCW}(\Phi \wedge \neg\Psi) \models \text{LCW}(\Phi)$. Introducing disjunction as well would make matters even worse. In general, answering a singleton LCW query, in the presence of negation and disjunction, is NP-hard.[2]

Theorem 2.1 (NP-hardness of LCW queries for unrestricted $\mathcal{L}$) *If $\mathcal{L}$ contains unrestricted* LCW *formulas and p is a single literal, then answering a query* LCW*(p) is NP-hard in the size of $\mathcal{L}$.*

Since our planner makes numerous singleton queries, we chose to sacrifice completeness in the interest of speed and restrict $\mathcal{L}$ to positive conjunctions.

2.4 Local Closed World Inference

Correctly answering LCW queries is not a simple matter of looking up assertions in a database. For instance, suppose the agent wants to establish whether it knows which files are in the `/kr94` directory, and it finds that it has LCW on the contents of *every* directory. Then, *a fortiori*, it knows which files are in `/kr94`. That is:

$$\text{LCW}(\texttt{in.dir}(f, d)) \models \text{LCW}(\texttt{in.dir}(f, \texttt{/kr94}))$$

---

[2] The proofs for all theorems are in Appendix A.

In general, we have:

Theorem 2.2 (Instantiation) *If Φ is a logical formula and θ is a substitution, then* $\textsc{lcw}(\Phi) \models \textsc{lcw}(\Phi\theta)$.

Moreover, LCW assertions can be combined to yield new ones. For example, if one knows all the group-readable files, and for each group-readable file, one knows whether that file is in `/kr94`, then one knows the set of group-readable files in `/kr94`. In general, if we know the contents of set A, and for each member of A, we know whether that member resides in another set B, then we know the intersection of sets A and B. More formally:

Theorem 2.3 (Composition) *If Φ and Ψ are logical formulas and* $\textsc{lcw}(\Phi)$ *and for all substitutions σ, we have that $\Phi\sigma \in \mathcal{S}$ implies* $\textsc{lcw}(\Psi\sigma)$*, then we can conclude* $\textsc{lcw}(\Phi \wedge \Psi)$.

Note that if the agent knows all the group-readable files, and it knows which files are located in `/kr94`, it follows that it knows the set of group-readable files in `/kr94`. This is a special case of the Composition Theorem, in which $\textsc{lcw}(\Psi)$ holds for all σ, but it's interesting in it's own right. In general, we have:

Corollary 2.4 (Conjunction) *If Φ and Ψ are logical formulas then*

$$\textsc{lcw}(\Phi) \wedge \textsc{lcw}(\Psi) \models \textsc{lcw}(\Phi \wedge \Psi).$$

The intuition behind this corollary is simple — if one knows the contents of two sets then one knows their intersection. Note that the converse is invalid. If one knows the group-readable files in `/kr94`, it does not follow that one knows *all* group-readable files. The rule $\textsc{lcw}(\Phi) \models \textsc{lcw}(\Phi \wedge \Psi)$ is also invalid. For instance, if one knows all the group-readable files, it does not follow that one knows exactly which of these files reside in `/kr94`.

Two additional observations regarding LCW are worth noting. Knowing whether an element is in a set is equivalent to knowing whether an element is *not* in the set:

Theorem 2.5 (Negation) *If* Φ *is a logical formula then* $\text{LCW}(\Phi) \models \text{LCW}(\neg\Phi)$.

Finally, if one knows the contents of two sets then one knows their union. More formally:

Theorem 2.6 (Disjunction) *If* Φ *and* Ψ *are logical formulas then*

$$\text{LCW}(\Phi) \wedge \text{LCW}(\Psi) \models \text{LCW}(\Phi \vee \Psi).$$

As we explain in the previous section, our representation of LCW sentences is restricted to positive conjunctions, so the above theorems are mainly of only theoretical interest. Given an LCW query containing disjunction and negation, an agent could use the previous two theorems to break it down into conjunctive LCW formulas, however, such an inference procedure might result in many false negatives. If the query succeeds, the original LCW formula is guaranteed to be true, but since but nothing can be concluded from a negative result.

2.5 Inference Method

We have discussed the semantics of LCW entailment in terms of $\mathcal{S}$, but since the agent has access only to the syntactic representations $\mathcal{L}$ and $\mathcal{M}$, we must describe inference in terms of these databases. As it turns out, the actual inference procedure directly corresponds to the Instantiation and Composition Theorems (Theorems 2.2 and 2.3). We can define the transitive closure of $\mathcal{L}$ using the following two rules.

1. **Instantiation Rule** If $\text{LCW}(\Phi) \in \mathcal{L}$ and θ is a substitution, then $\mathcal{L}' \leftarrow \mathcal{L} \cup \{\text{LCW}(\Phi\theta)\}$.

2. **Composition Rule** If $\text{LCW}(\Phi) \in \mathcal{L}$ and for all ground substitutions θ $(\Phi\theta \in \mathcal{M} \Rightarrow \text{LCW}(\Psi\theta) \in \mathcal{L})$ then $\mathcal{L}' \leftarrow \mathcal{L} \cup \{\text{LCW}(\Phi \wedge \Psi)\}$

Given the direct correspondence between these two rules and Theorems 2.2 and 2.3, this inference process (denoted $\vdash$) is clearly sound. Unfortunately, however, the inference rules are incomplete.

Theorem 2.7 (Incompleteness) *Let $\mathcal{M}$ be a set of consistent ground literals and let $\mathcal{L}$ be a set of positive conjunctive* LCW *formulas. There may exist an* LCW *formula* $\text{LCW}(\Phi)$ *that logically follows from $\mathcal{L}$ and $\mathcal{M}$, but which is not in the transitive closure of $\mathcal{L}$ given the Instantiation and Composition rules.*

Fortunately, the incompleteness of these inference rules is not a problem in practice. Section 7.2.2 provides an empirical demonstration that they miss substantially fewer than 1% of the LCW inferences requested during Softbot operation.

Note that maintaining an explicit transitive closure of $\mathcal{L}$ is impractical. For each LCW formula in $\mathcal{L}$, the Instantiation Rule alone generates a number of new LCW formulas that is polynomial in the number of objects in the universe. Given finite memory resources, we choose instead to compute the closure lazily, by performing the necessary inference during queries. Figure 2.2 shows the inference algorithm.[3] Since the correctness of this backward-chaining algorithm is less obvious than that of the inference rules used to define the transitive closure, we prove soundness formally.

Theorem 2.8 (Soundness) *Let $\mathcal{M}$ be a set of consistent ground literals and let $\mathcal{L}$ be a set of positive, conjunctive* LCW *formulas such that $\mathcal{M}$ and $\mathcal{L}$ form a conservative representation of $\mathcal{S}$. If* `QueryLCW(Φ, M, L)` *returns* `T` *then* $\text{LCW}(\Phi)$.

In the worst case, `QueryLCW` has to consider all possible decompositions, which is exponential in the number of conjuncts in the query.

[3] This algorithm omits the conjunction rule, since it is subsumed by the composition rule. In practice, the conjunction rule is applied when applicable, since it is more efficient.

function QueryLCW(Φ, $\mathcal{M}$, $\mathcal{L}$**)**: Boolean

1 QLCW*(Φ, {}, $\mathcal{M}$, $\mathcal{L}$)

function QLCW*(Φ, *Matches*, $\mathcal{M}$, $\mathcal{L}$**)**: Boolean

1 **if** $\Phi = \{\}$ **then return** `T`

2 **else if** Φ is ground and $\exists \varphi \in \Phi, \neg\varphi \in \mathcal{M}$ or $\forall \varphi \in \Phi, \varphi \in \mathcal{M}$

  then return `T`

3 **else for** C such that LCW(C) $\in \mathcal{L}$ **do**

4   **for** $\Phi' \subseteq \Phi$ such that $\exists\theta, \Phi' \subseteq C\theta$ **do**

5   **if** $(C\theta - \Phi') \subseteq Matches \wedge |\Phi'| > 0$ **then**

6   **if** $\forall\sigma \in$ ConjMatch($Matches \cup \Phi'$, $\mathcal{M}$)

          QLCW*$((\Phi - \Phi')\sigma, (\Phi')\sigma, \mathcal{M}, \mathcal{L})$

          then return `T`

7 **return** `F`

Figure 2.2: The `QueryLCW` algorithm determines whether a conjunctive LCW statement follows from the agent's beliefs as encoded in terms of the $\mathcal{M}$ and $\mathcal{L}$ databases. Since $\mathcal{L}$ is restricted to positive conjunctions, LCW inference is reduced to the problem of matching a conjunctive LCW query against a database of conjunctive LCW assertions. A successful match occurs when repeated applications of the Composition Rule (line 6) decompose the query into sub-conjunctions, which are directly satisfied by the Instantiation Rule applied to $\mathcal{L}$ (line 4) or reduced to ground formulas and found in $\mathcal{M}$ (line 2). Note that unlike `Query`, the `QueryLCW` algorithm allows variables in its Φ input. `QueryLCW` calls the QLCW* helper function which calls ConjMatch in turn. ConjMatch(C, $\mathcal{M}$) performs a standard conjunctive match, returning all bindings θ, such that $\mathcal{M} \models C\theta$. The variable $Matches$ represents all conjuncts of the original query that have so far been matched by some LCW formula. The query is satisfied when all conjuncts have been matched. Matching against a conjunct multiple times is permitted, which is why $\Phi' \subseteq C\theta$ in line 4 and $(C\theta - \Phi') \subseteq Matches$ in line 5. Line 5 guarantees that progress is made in each recursive invocation, so the depth of the recursion is bounded by the number of conjuncts in Φ. We use the notation $\Phi - \Phi'$ to denote the conjunction Φ with conjuncts in Φ' omitted.

Theorem 2.9 (Complexity of `QueryLCW`) *Let Φ be a positive conjunction with c conjuncts, let $\mathcal{M}$ be a set of ground literals, and let $\mathcal{L}$ be a set of positive conjunctive* LCW *sentences. In the worst case,* `QueryLCW(`Φ`, `$\mathcal{M}$`, `$\mathcal{L}$`)` *may require $O(|\mathcal{L}|^c|\mathcal{M}|^c)$ time.*

For our purposes, the number conjuncts in an LCW query is bounded by the planning domain theory (*i.e.*, the set of available actions) used by the agent. In our softbot's domain (see Appendix B) for example, LCW queries are typically short ($c \leq 2$) and never greater than 4. As a result, the worst case complexity is polynomial in the size of $\mathcal{L}$ and $\mathcal{M}$. With the aid of standard indexing techniques, this yields extremely fast LCW inference in practice. In our experiments, LCW queries averaged about 2 milliseconds (see Section 7.2 for the details).

2.6 Summary

We have presented a representation of incomplete information, which is described semantically in terms of the situation calculus and a possible-worlds interpretation of knowledge. However, for the sake of tractability, this representation uses three-valued logic and LCW formulas to represent the agent's locally complete information. We have presented sound inference mechanisms for this LCW knowledge, and have shown that the inference can be done in polynomial time.

In the next chapter, we discuss changes to this knowledge brought about by the execution of actions. We also discuss goals, including goals of acquiring information.

Chapter 3

ACTION AND CHANGE

One of the stumbling blocks to past research in planning with incomplete information has been inadequate or imprecisely defined languages for representing information goals and sensing actions. Many researchers have devised formalisms for reasoning about knowledge and action [63, 64, 65, 17, 9, 80, 51], but those languages are too expressive to be used in practical planning algorithms. UWL [27] offered a more tractable representation (based on STRIPS) that was tailored to current planning technology, but as Levesque [51] observes, the semantics of UWL are unclear — the definitions were made relative to a specific planning algorithm. In our efforts to define a semantics for UWL, we determined that UWL confused information goals with maintenance goals, and conflated knowledge goals with knowledge preconditions. Furthermore, years of experience with UWL convinced us that it wasn't expressive enough to fully handle the real-world domains (*e.g.*, UNIX and the Internet) for which it was intended. Since UWL didn't support universal quantification or conditional effects, it could not correctly represent the UNIX command `ls`, which lists all files in a directory, or `rm *`, which deletes all writable files.

In this chapter, we present the action representation language SADL,[1] which combines ideas from UWL with those from Pednault's ADL [71, 69]. Just as ADL marked the "middle ground" on the tractability spectrum between STRIPS and the situation calculus, SADL offers an advantageous combination of expressiveness and efficiency. Since SADL supports universally quantified information goals and universally quanti-

[1] SADL (pronounced "Saddle") stands for "Sensory Action Description Language."

fied, conditional, observational effects, it is expressive enough to represent hundreds of UNIX and Internet commands (see Appendix B for some examples). Indeed, four years of painful experience writing and debugging the Internet Softbot [26] knowledge base forced us to uncover and remedy some subtle confusions about information goals:

- In a dynamic world, knowledge goals are inherently *temporal* — If proposition P is true at one time point and false in another, which time point do we mean when we ask about P's truth value? Since UWL has limited provision to make temporal distinctions, it cannot encode an important class of goals. In particular, UWL cannot express goals that require causal change to attributes used to designate objects, *e.g.*, "Rename the file `paper.tex` to `kr.tex`." (See Sections 3.2.2 and 3.2.3 for the SADL solution)

- We identify a large class of domains, called *Knowledge-free Markov domains*, and argue that actions in these domains are best encoded *without knowledge preconditions*. The multiagent scenarios that inspired Moore, Morgenstern, and others are not knowledge-free Markov, but UNIX and much of the Internet are. While SADL discourages knowledge *preconditions* it recognizes the need for knowledge *subgoals*. (Section 3.2.6 elaborates).

Section 3.1 discusses how the semantics of SADL are described using situation calculus. Section 3.2 describes problems with the UWL formulation of knowledge goals and presents the SADL solution. In Section 3.3, we discuss observational effects of actions, and causal effects, which can decrease the agent's knowledge about the world. We also demonstrate the representational adequacy of SADL by presenting encodings for 50 of the UNIX commands (see Appendix B). In Section 3.4 we discuss temporal projection in SADL. In Section 3.5 we discuss how executing actions affects LCW knowledge.

3.1 *The Situation Calculus*

Although the semantics of UWL was defined procedurally [27], we provide SADL's semantics in terms of the situation calculus (discussed in Section 2.1). All state changes are assumed to result from the execution of actions. The special function DO is used to describe these changes: $\mathrm{DO}(a, s)$ returns the situation resulting from executing action a in situation s. We use $\{a\}_1^n$ to represent the sequence of actions a_1; a_2; ...; a_n. $\mathrm{DO}(\{a\}_1^n, s)$ denotes nested application $\mathrm{DO}(a_n, \mathrm{DO}(a_{n-1}, \ldots, \mathrm{DO}(a_1, s)))$, *i.e.*, the result of executing the entire sequence, starting in situation s. We use s_0 to represent the initial situation and we use s_n as a shorthand for $\mathrm{DO}(\{a\}_1^n, s_0)$. Our formulation of SADL is based on Scherl and Levesque's [80] solution to the frame problem for knowledge-producing actions. We adopt their Completeness Assumption, and their formulation of incomplete knowledge (as discussed in Chapter 2), and thus their results (*i.e.*, the persistence of knowledge and of ignorance) hold for us as well.

The Completeness Assumption is essentially the STRIPS assumption — that is, all changes produced by executing an action are listed in the action's effects, so anything else can be assumed to stay the same. This holds for the K operator as well as ordinary fluents, giving us persistence of knowledge. The Completeness Assumption, like the STRIPS assumption, is used to solve the frame problem, but in the situation calculus, this assumption needs to be axiomatized explicitly. We discuss how the frame problem is solved in the framework of the situation calculus in Section 3.4.

Another standard assumption we adopt is the Unique Names Assumption. That is, for all situations s_1, s_2 and all actions a_1, a_2, $\mathrm{DO}(a_1, s_1) = \mathrm{DO}(a_2, s_2)$ if and only if $a_1 = a_2$ and $s_1 = s_2$. In other words, all situations with unique histories are unique, even if the conditions that hold in them are the same. This is reasonable, since situations with different past histories are different by virtue of their histories.

We face a slight complication when we want to say something about SADL goals and effects using the situation calculus, since SADL expressions are not objects in

the situation calculus. For this reason, we introduce two predicates that take SADL expressions as arguments: We define ACHV$(G, s_0, \{a\}_1^n)$ to mean that the SADL goal G is achieved in the situation resulting from executing plan $\{a\}_1^n$ in situation s_0. EFF(E, a, s) means E becomes true after action a is executed in situation s

3.2 Goals

In UWL, preconditions and goals were limited to conjunctions of literals, each annotated with one of three tags: **satisfy**, **hands-off**, and **find-out**. The SADL action language is based on UWL, but uses a different set of annotations: **satisfy**, **hands-off**, and **initially**, which provide a cleaner semantics for information goals and greater expressive power; additionally, SADL uses unannotated literals to designate preconditions that don't depend on the agent's knowledge. Furthermore, SADL supports universal quantification and conditional effects, both of which have interesting ramifications in the context of incomplete information. We proceed by reviewing UWL, uncovering some confusions, presenting the SADL solution, and sketching the formal semantics.

In UWL (and in SADL) individual literals have truth values expressed in a three-valued logic: `T`, `F`, `U` (unknown). Free variables are implicitly existentially quantified, and the quantifier takes the widest possible scope.[2] For example, **satisfy**(in.dir (f, `tex`), `T`)[3] means "Find a file in directory `tex`." Truth values can also be represented by variables. For example, **satisfy**(in.dir (`myfile`, `tex`), tv) means "Find out whether or not `myfile` is in `tex`."

---

[2] Explicit quantifiers can be used to indicate a narrower scope.

[3] For notational convenience, an omitted truth value defaults to `T`, so this could be rewritten as **satisfy**(in.dir (f, `tex`)). We use this shorthand in the remainder of the paper.

3.2.1 Satisfaction and Maintenance Goals

The goal **satisfy**(P) indicates a traditional goal (as in ADL [69]): achieve P by whatever means possible. In the presence of incomplete information, we make the further requirement that the agent knows that P is true. Recall that $\text{ACHV}(G, s_0, \{a\}_1^n)$ means that goal G is achieved in the situation resulting from executing plan $\{a\}_1^n$ in situation s_0; since we assume the agent's knowledge is correct, it is sufficient to state that the agent knows P:

$$\text{ACHV}(\textbf{satisfy}(P, \mathtt{T}), s_0, \{a\}_1^n) \equiv \text{KNOW}(P, s_n) \tag{3.1}$$

$$\text{ACHV}(\textbf{satisfy}(P, \mathtt{F}), s_0, \{a\}_1^n) \equiv \text{KNOW}(\neg P, s_n) \tag{3.2}$$

$$\begin{aligned} &\text{ACHV}(\textbf{satisfy}(P, tv), s_0, \{a\}_1^n) \equiv \\ &(\text{KNOW}(P, s_n) \wedge tv = \mathtt{T}) \vee (\text{KNOW}(\neg P, s_n) \wedge tv = \mathtt{F}) \end{aligned} \tag{3.3}$$

Note that when given an (existentially quantified) variable as truth value, a **satisfy** goal requires that the agent learn whether the proposition is true or false (which could be achieved by making it true or false). The variable tv in Equation 3.3 serves as a proposition that has the same truth value that P has in situation s_n. The reason for including it is to ensure that, given a goal such as **satisfy**$(P, tv) \wedge$ **satisfy**(Q, tv), we get the correct interpretation that P and Q must have the same truth value in situation s_n. Because tv is essentially just a proposition, in the context of boolean formulas, we will abbreviate $tv = \mathtt{T}$ as tv and we will write $tv = \mathtt{F}$ as $\neg tv$. If tv only appears once in a formula, its value is unimportant, and Equation 3.3 can be simplified to:

$$\text{ACHV}(\textbf{satisfy}(P, tv), s_0, \{a\}_1^n) \equiv \begin{array}{l} \text{KNOW}(P, s_n) \vee \\ \text{KNOW}(\neg P, s_n) \end{array} \tag{3.4}$$

In other words, in situation s_n, the agent knows whether or not P is true.

The **hands-off** annotation indicates a maintenance goal that prohibits the agent from changing the fluent in question. As a notational shorthand, we use ORIG_n

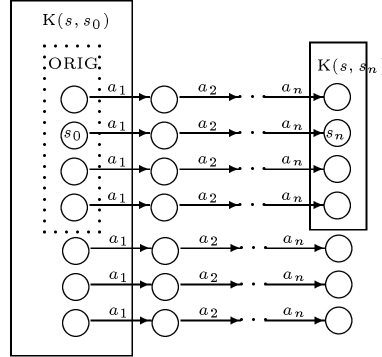

Figure 3.1: The region surrounded by dotted lines represents the set ORIG_n, the set of states indistinguishable from s_0, based on the agent's knowledge in state s_n. ORIG_n is a subset of $\{s \mid \mathrm{K}(s, s_0)\}$, the states that were consistent with the agent's knowledge in s_0, since the agent has learned more about what originally held, but has not forgotten anything it knew originally.

(Figure 3.1) to represent the agent's knowledge in s_n *about the past situation* s_0, *i.e.*, the set of situations indistinguishable from s_0 after execution of the plan: $\mathrm{ORIG}_n = \{s \mid \mathrm{K}(\mathrm{DO}(\{a\}_1^n, s), \mathrm{DO}(\{a\}_1^n, s_0))\}$. The definition of **hands-off** is then:

$$\mathrm{ACHV}(\textbf{hands-off}(P), s_0, \{a\}_1^n) \equiv \bigwedge_{i=1}^{n} \forall s \in \mathrm{ORIG}_i \; [P(\mathrm{DO}(\{a\}_1^i, s) \Leftrightarrow P(s)] \quad (3.5)$$

Thus, the definition of **hands-off** requires that P not change value during execution of the plan. [27] noted that together, **satisfy** + **hands-off** can be used to indicate a "look but don't touch" goal: the agent may sense the fluent's value, but is forbidden to change it. While **hands-off** goals are clearly useful, we argue that they are an overly restrictive way of specifying *knowledge* goals. In particular, they outlaw changing the value of a fluent after it has been sensed.

3.2.2 Knowledge Goals are Inherently Temporal

Before explaining the SADL approach to knowledge goals, we discuss the UWL **find-out** annotation. **find-out** is problematic because the original definition was in terms of a

particular planning algorithm [27]. The motivation for **find-out** was the existence of goals for which **hands-off** is too restrictive, but **satisfy** alone is too permissive. For example, given the goal "Tell me what files are in directory `tex`," executing `rm tex/*` and reporting "None" would clearly be inappropriate. But what about the conjunctive goal "Free up some disk space and tell me what files are in directory `tex`"? In this case *excluding* the `rm` seems inappropriate, since it may be necessary in service of freeing disk space. Yet the knowledge that the directory is now empty is relevant to the information goal. Proponents of **find-out** argued that `rm` was unacceptable for the first goal, but acceptable in service of the conjunction [27]. We contend that this definition is unclear and unacceptable; a plan that satisfies the conjunction $A \wedge B$ should also be a solution to A.

While the examples used to justify the original **find-out** definition are evocative, their persuasive powers stem from *ambiguity.* At what *time point* do we wish to know the directory contents? Before freeing disk space, afterward, or in between? Since fluents are always changing, a general information goal requires two temporal arguments: the time a fluent is sensed, and the time the sensed value is to be reported. *E.g.*, one can ask "Who was president in 1883," or "Tell me tomorrow who was president today."

Since planning with an explicit temporal representation is slow, our quest for the "middle ground" along the expressiveness / tractability spectrum demands a minimal notion of time that captures most common goals. We limit consideration to two time points: the time when a goal is given to the agent, and the time the agent gives its reply. Note that **satisfy**(P, tv) (Equation 3.3) allows one to specify the goal of knowing P's truth value at this latter time point. To specify the goal of sensing a fluent at the time the goal is given, we introduce the annotation **initially** .

$$\text{ACHV}(\textbf{initially}(P, tv), s_0, \{a\}_1^n) \equiv \forall s \in \text{ORIG}_n(P(s) \Leftrightarrow tv) \quad (3.6)$$

The definition of **initially** states that when the agent has finished executing the

plan, it will know whether P was true or false when it started. **initially**(P) is not achievable by an action that changes the fluent P, since such an action only obscures the initial value of P. However, changing P after determining its initial value is fine. By combining **initially** with **satisfy** we can express "tidiness" goals (modify P at will, but restore its initial value by plan's end) [87]. Furthermore, we can express goals such as "Find the the file currently named `paper.tex`, and rename it to `kr.tex`," which are impossible to express in UWL. Since UWL can't make temporal distinctions, there is no way to ask for the past value of a fluent without also requiring that the fluent have the same value when the reply is given, so any goal of the form "Find some x such that $P(x)$, and make $P(x)$ false" is inexpressible in UWL.

3.2.3 Universally Quantified Goals

When defining universally quantified goals, one must again be specific with respect to time points: does the designator specifying the Herbrand universe refer to s_0 or s_n? Since SADL allows an arbitrary goal description to be used to scope a universally quantified goal, one can specify a wide range of requests. For example, suppose an agent is given the goal of seeing to it that all files in directory `tex` are compressed. What plans satisfy the goal? It depends on what the request really means. In SADL, one can write one of the following precise versions, thus eliminating the ambiguity.

1. Ensure that all files that were initially in `tex` end up being compressed: $\forall f$ **initially**(in.dir (f, `tex`)) $\Rightarrow$ **satisfy**(compressed (f)). Executing `compress tex/*` solves this goal, as does executing `mv tex/* temp` then `compress temp/*`.

2. Ensure that all files that end up in `tex` end up being compressed: $\forall f$ **satisfy**(in.dir (f, `tex`)) $\Rightarrow$ **satisfy**(compressed (f)). Executing `compress tex/*` solves this goal, but so does `rm tex/*`!

3. Determine if all files initially in `tex` were initially compressed: $\forall f$ **initially**(in.dir (f, `tex`)) $\Rightarrow$ **initially**(compressed (f)).

4. Determine if all files, in `tex` at the end of execution, were initially compressed: $\forall f$ **satisfy**(in.dir (f, `tex`)) $\Rightarrow$ **initially**(compressed (f)).

The first example seems the most likely interpretation of the goal in this case, but it still leaves something to be desired, since the user may not want the files moved from `tex`. We can easily state the additional requirement that the files not be moved (**hands-off**(in.dir (f, `tex`))), or that they be returned to `tex` by the end (**satisfy**(in.dir (f, `tex`))). We should be careful not to make goals overly restrictive, though. If the desire is that the agent should fail if there's no way to compress the files without moving them, then adding such restrictions is correct. If the desire is merely that the agent should avoid moving the files unnecessarily, then we want the original solution, with some background preference to minimize unnecessary changes. Such background preferences could be expressed in terms of a utility function over world states [76], a measure of plan quality [74, 88], or an explicit notion of harm [87].

Note that even if we decide to forbid moving the files from `tex`, there are still other actions, such as deleting all the files in `important/papers`, or sending threatening email to `president@whitehouse.gov` that haven't been excluded. This is a general problem with satisficing plans: anything goes as long as the goal is achieved. Specifying all the undesired outcomes with every goal would be tedious and error-prone. A better solution is to separate the criteria of goal satisfaction from background preferences, as is done in [89, 36, 87].

Given the appropriate annotations on fluents, which provide temporal information, the semantics of $\forall$ goals is straightforward:

$$\text{ACHV}(\forall \vec{x}.P, s_0, \{a\}_1^n) \equiv \forall \vec{x}.\text{ACHV}(P, s_0, \{a\}_1^n) \tag{3.7}$$

$$\text{ACHV}(P \Rightarrow Q, s_0, \{a\}_1^n) \equiv \begin{array}{l} \text{ACHV}(P, s_0, \{a\}_1^n) \Rightarrow \\ \text{ACHV}(Q, s_0, \{a\}_1^n) \end{array} \tag{3.8}$$

Logical operators such as $\wedge$, $\vee$, and $\exists$ follow the same form as above.

3.2.4 LCW *Goals*

As we discuss in Section 5.2.5, one way of satisfying universally quantified goals involves subgoaling on LCW goals. Our discussion of LCW in Section 2.2 is incomplete because, as with universally quantified goals, we must also specify the time point at which we want complete information. We augment our earlier definition of LCW formulas by allowing individual conjuncts to be labeled with **initially**, indicating that the time point over which we want complete information is the time when the goal was given to the agent. This change does not effect the definitions or algorithms given earlier – it merely adds another kind of term. conjuncts of an LCW goal tagged with **initially** are sent off to a separate $\mathcal{L}$ database, which describes LCW over the initial state, but the inference procedures are the same. Updates to this other $\mathcal{L}$ database are discussed in Section 3.5.5.

LCW goals and LCW effects are part of the SADL language, but they typically don't appear directly in user-specified goals or actions. Rather, they are generated automatically by the planner, as discussed in Sections 3.5 and 5.2.5.

3.2.5 Variables, Types, and Predicates

SADL is strongly typed. Every variable has a type, which may be inferred from its context rather than being declared explicitly. The definition of every predicate includes a type for each argument. For example, the in.dir predicate requires its first argument to be a file and its second argument to be a directory (which is a kind of file). Predicate arguments also have a specified *cardinality*. For example, the pathname predicate is functional in both arguments. Given a file, there is exactly one pathname, and vice versa. The filename relation is functional in only one of its arguments (each file has exactly one name, but different files can have the same name). The string.in.file relation, which states that a given string appears somewhere in a given file, is not functional in any argument.

Formally speaking, a type is merely a unary predicate. The requirement that all arguments be typed may be regarded as a discipline that is enforced by the planner. The type declarations themselves may be regarded as logical inference rules that relate predicates to the types of their arguments, and types to their subtypes. Cardinality information can also be regarded in terms of inference rules. [4] Types and cardinality in SADL thus are a notational convenience, and have no real impact on the semantics of the language.

3.2.6 Knowledge Preconditions Considered Harmful

Moore [63] identified two kinds of knowledge preconditions an agent must satisfy in order to execute an action in support of some proposition P: First, the agent must know a rigid designator (*i.e.*, an unambiguous, executable description) of the action. Second, the agent must know that executing the action will in fact achieve P. Subsequent work, *e.g.* [64], generalized this framework to handle scenarios where multiple agents reasoned about each other's knowledge.

In the interest of tractability, we take a much narrower view, assuming away Moore's first type of knowledge precondition and refuting the need for his second type. Our argument occupies the remainder of this section, but the summary is that there is a large class of domains, which we call *Knowledge-free Markov domains*, for which actions are best encoded without knowledge preconditions. While the multiagent scenarios considered by Moore and Morgenstern are not knowledge-free Markov, UNIX and much of the Internet are.

We start the argument by assuming away Moore's first type of knowledge precondition. We define actions as programs that can be executed by a robot or softbot, without the need for further reasoning. In this view, *all* actions are rigid designators. `dial` (combination(`safe`)) is not an admissible action, but `dial(31-24-15)` is. Lifted

[4] Note, however, that these are the *only* inference rules supported by the planner, and that since type declarations are static and subtype relations are hierarchical, this inference is of an extremely simple nature.

action schemas, *e.g.* `dial(`x`)`, are not rigid designators, but it is easy to produce one by substituting a constant for x. Thus Moore's first type of knowledge precondition vanishes.

Moore's second type of knowledge precondition presupposes that an action in a plan must provably succeed in achieving a desired goal. This is a standard assumption in classical planning, but is overly restrictive given incomplete information about the world; enforcing this assumption by adding knowledge preconditions to actions is inappropriate. For example, if knowledge of the safe's combination is a precondition of the `dial` action, then it becomes impossible for a planner to solve the goal "find out whether the combination is 31-24-15" by dialing that number, since before executing the `dial` action, it will need to satisfy that action's precondition of finding out whether 31-24-15 is the right combination![5]

On the other hand, it is often necessary for an agent to plan to obtain information, such as the combination of a safe, either to reduce search or to avoid dangerous mistakes. These knowledge *subgoals*,[6] naturally, have a temporal component, but the only time point of interest is the moment the action is executed. For example, the goal of knowing the safe's combination could be satisfied by watching another agent open the safe, but it might also be satisfied by changing the combination to some known value (for instance, at some earlier time when the safe is open).

We say that an action is *knowledge-free Markov (KFM)* if its effects depend only

---

[5] Note that eliminating the knowledge precondition from the `dial` action also allows the unhurried agent to devise a plan to enumerate the possible combinations until it finds one that works. Indeed, the Internet Softbot [26] follows an analogous strategy when directed to find a particular user, file or a web page, whose location is unknown. If `finger` and `ls` included knowledge preconditions, then the actions would be useless for locating users and files.

[6] It is commonplace in the planning literature to conflate goals and preconditions. This is undoubtably because an agent naturally adopts the preconditions of an action (or an effect) as its goals when it wants to use the action for some purpose. This conflation is encouraged by planning algorithms (including PUCCINI) that represent high-level goals as preconditions of a "goal step." We believe this trend has led to some confusion, so we make a firm distinction between preconditions, which are descriptions of an action (and hence of the world), and goals, which are part of the agent's mental state. The agent is free to adopt action preconditions as goals, but the two are very different concepts

on the state of the world (and *not* on that agent's knowledge about the world) at the time of execution. Note that simple mechanical and software systems are naturally encoded as KFM, while multiagent systems are typically *not*, because it is useful to endow one's model of another agent with state (*i.e.*, I know that Bill knew ...). The abstract actions used by HTN planners are also not KFM, since these are more plans than they are actions, and depend on the agent's knowledge to expand appropriately.

If all actions in a domain are KFM, then all knowledge sub-goals will be of the same form: 1) The agent needs to know the value of some fluent at the time the action is to be executed, and 2) it doesn't matter if the agent affects the fluent while obtaining its value.[7] These requirements for knowledge sub-goals are met by the SADL definition of **satisfy** (Equation 3.3),[8] if we regard the action sequence $\{a\}_1^n$ as a plan to achieve the preconditions of action a_{n+1}. In Section 5.2.7, we discuss how the PUCCINI planner is able to adopt goals of acquiring information, without making that a prerequisite for executing actions.

The knowledge-free Markov assumption for actions yields a substantially simpler representation of change than those defined by Moore and Morgenstern. While their theories are more appropriate for complex, multi-agent domains, SADL gains tractability while retaining enough expressive power to model many important domains.

---

[7] The reader may object that (non-rigid) indexical references could appear as preconditions to actions. For example, suppose that running Netscape requires that *the* file `netscape.bookmarks` be in a given directory. It is not sufficient that *a* file of that name be there, because renaming `paper.tex` to `netscape.bookmarks` would cause Netscape to fail. But this example makes it clear that the proposed preconditions of Netscape are simply under-specified. They should be "The directory contains a file named `netscape.bookmarks`, which is a valid bookmarks file, and ..." This is just the qualification problem [59] in disguise. Granted, it will usually be impossible (or undesirable) to model all such preconditions.

[8] A justification that might be given for **initially** or **hands-off** preconditions is to minimize destructive actions used by an agent to satisfy a goal (*i.e.* don't use `mv` to find out the name of a file). We agree on the need for reasoning about *plan quality*, but an accurate theory of action should distinguish action preconditions from *user preferences.*

3.3 Effects

Following UWL [27], SADL divides effects into those that change the world, annotated by **cause**, and those that merely report on the state of the world, annotated by **observe**. Because it lacked universal quantification, UWL couldn't even correctly model UNIX `ls`. SADL goes beyond UWL by allowing both observational and causal effects to have preconditions and universal quantification.

3.3.1 Observational Effects

Executing actions with observational effects assigns values to *runtime* variables that appear in those effects. By using a runtime variable as a parameter to a later action (or to control contingent execution), information gathered by one action can affect the agent's subsequent behavior. Inside an effect, runtime variables (syntactically identified with a leading an exclamation point, *e.g.* *!tv*) can appear as terms or as truth values. For example, `ping twain` has the effect of **observe**(machine.alive(`twain`), *!tv*), *i.e.* determining whether it is true or false that the machine named `twain` is alive, and `wc myfile` has the effect **observe**(word.count(`myfile`, *!word*)), *i.e.* determining the number of words in `myfile`.

Before we define individual effects, we discuss what it means to execute an action, with all its effects. Recall that $\text{EFF}(E, a, s)$ means E becomes true after action a is executed in s. Let π_a be the precondition of action a, and let ε_a be the effects. An action's effects will only be realized if the action is executed when its preconditions are satisfied. Furthermore, the agent always knows when it executes an action, and it knows the effects of that action. Following Moore [63]:

$$\begin{aligned} &\forall s.\text{ACHV}(\pi_a, s, \{\}) \Rightarrow \forall s''.[K(s'', \text{DO}(a, s)) \Leftrightarrow \\ &\exists s'.\text{K}(s', s) \wedge s'' = \text{DO}(a, s') \wedge \text{EFF}(\varepsilon_a, a, s)] \end{aligned} \tag{3.9}$$

The agent's knowing the effects of a doesn't imply that those effects are always certain. As we discuss in Section 3.3.3, actions with conditional effects can result in uncertainty.

We now define the semantics of **observe** in terms of primitive situation calculus expressions:

$$\text{EFF}(\textbf{observe}(P, \texttt{T}), a, s) \equiv \forall s'(\text{K}(s', \text{DO}(a, s)) \Rightarrow$$
$$\exists s_i.\text{K}(s_i, s) \wedge s' = \text{DO}(a, s_i) \wedge P(s_i)) \tag{3.10}$$
$$\text{EFF}(\textbf{observe}(P, tv), a, s) \equiv \forall s'(\text{K}(s', \text{DO}(a, s)) \Rightarrow$$
$$\exists s_i.\text{K}(s_i, s) \wedge s' = \text{DO}(a, s_i) \wedge (P(s_i) \Leftrightarrow tv)) \tag{3.11}$$

The situations s' above refers to all situations that the agent considers possible after executing a from situation s. Because the agent's knowledge is correct, $\text{DO}(a, s)$ will be one such situation. The s_i refers to the situation immediately preceding s', *i.e.*, the situation that would have held prior to executing a if the situation afterward were in fact s'. Thus the set of situations s_i consist of all the *previous* states that the agent considers to have been possible, *after* executing a. Since the agent has performed an observation, the set of all such situations s_i will be a subset of situations that were considered possible in situation s. Since the agent's knowledge is consistent, one of the situations s_i will be s itself.

Note that if we substitute $\text{DO}(a, s)$ for s' in Equation 3.11, we find that $P(s) \Leftrightarrow tv$. Since tv has the same truth value as $P(s)$, Equation 3.11 gives us $P(s_i) \Leftrightarrow P(s)$, meaning that the agent knows, after executing a, whether or not P was true in situation s.

In other words, if action a has an effect $\textbf{observe}(P, tv)$ and is executed in situation s, then in the resulting situation, the agent knows what value P had in s. For example, if in s the agent observes that the sky is blue, we would say that in situation $s' = \text{DO}(\texttt{look}, s)$, the agent knows that the sky was blue in situation s. The double

use of the K operator in Equations 3.9 and 3.10/3.11 is a trifle redundant given only a single observational effect. Indeed, if we assume positive introspection (*i.e.* K is transitive), as in the modal logic S4 [37], the resulting equation can be greatly simplified. However, in more complex effects, we wish to distinguish between the agent knowing that the effect as a whole took place, and knowing the value of a single fluent.

SADL supports universally quantified run-time variables. By nesting universal and existential quantifiers, SADL can model powerful sensory actions that provide several pieces of information about an unbounded number of objects. For example, `ls -a`, (Figure 1.2), reports several facts about each file in the current directory. The universal quantifier indicates that, at execution time, information will be provided about *all* files *!f* that are in directory *d*. Since the value of *!f* is observed, quantification uses a run-time variable. The nested existential quantifier denotes that each file has a *distinct* filename and pathname. The conditional **when** restricts the files sensed to those in directory *d*. It may seem odd that the in.dir relation appears in two places, but as we shall explain, the first use of in.dir refers to the actual situation s, whereas the second refers to the agent's knowledge (*i.e.*, all possible situations). Figure 3.2 shows an EBNF description of the SADL language.

It is useful to note that, after executing `ls -a tex`, the agent not only knows all files in `tex`; it knows that it knows all files (*i.e.*, it has LCW on the contents of `tex`). Because of the $\forall$ in the effects of `ls`, and since it knows the effects of `ls`, the agent can infer closed-world knowledge. Such inference would be costly if it were done using first-order theorem-proving in the situation calculus. We have devised efficient algorithms for doing this reasoning, which we describe in Section 3.5.

The translation of $\forall$ effects into the situation calculus is straightforward (other logical operators follow the same form):

$$\mathrm{EFF}(\forall \vec{x}.E, a, s) \equiv \forall \vec{x}.\mathrm{EFF}(E, a, s) \tag{3.12}$$

This definition of ∀ effects may seem anticlimactic. The magic, however, stems from the way in which **when** introduces secondary preconditions, restricting the universe of discourse to a (possibly) finite set, and indicating precisely the range of the quantifier.

3.3.2 Conditional Effects

An *effect precondition*, also known as a secondary precondition, defines the conditions under which action execution will achieve that effect. Unlike action preconditions, effect preconditions need not be true for the action to be executed. If p is the effect precondition of effect e, then the resulting conditional effect is defined as:

$$\text{EFF}(\textbf{when}\ p\ \ e), a, s) \equiv p(s) \Rightarrow \text{EFF}(e, a, s)$$

Note the use $p(s)$ on the left side of the $\Rightarrow$, instead of some expression involving **satisfy**, **hands-off** or **initially**. We don't use these annotations because the **when** preconditions are not like the goals we have discussed so far. Since they need to hold, if at all, when the action is executed, they are different from **initially** preconditions. But **satisfy** requires that the agent know that the condition is true, which would lead to the faulty conclusion that the effect only occurs if the agent *knows* that the effect preconditions hold. So we omit the annotation, to indicate that the conditions must hold at the time of execution, with or without knowledge of the agent.

This ensures that whether the effects occur depends only on the state of the world. It also makes it clear what is being quantified over in `ls`: The files *really* in d, at the time of execution.

3.3.3 Uncertain Effects

In some cases, executing actions with causal effects can decrease the agent's knowledge about the world. SADL provides two ways of encoding these actions: as conditional

effects whose precondition is unknown, or by explicitly specifying the `U` truth value. As an example of the former, executing `rm tex/*` deletes all writable files in `tex`; if the agent doesn't know which files are writable, then it won't know which files remain in `tex` even if it knew the contents before executing the action. As an example of explicit creation of uncertainty, we encode `compress myfile` with the effect $\forall n$ **cause** `(size (myfile,` n`), U)`.[9] We define causal effects for `T`, `F` and `U` truth values as follows:

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{T}), a, s) \equiv P(\mathrm{DO}(a, s)) \tag{3.13}$$

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{F}), a, s) \equiv \neg P(\mathrm{DO}(a, s)) \tag{3.14}$$

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{U}), a, s) \equiv$$

$$Unk_P(a, \mathrm{DO}(a, s)) \Leftrightarrow P(\mathrm{DO}(a, s)) \tag{3.15}$$

where, Unk_P is a predicate such that

$$\neg KNOW(Unk_P(a), \mathrm{DO}(a, s)) \wedge$$

$$\neg KNOW(\neg Unk_P(a), \mathrm{DO}(a, s)) \tag{3.16}$$

In other words, we represent an uncertain effect as a deterministic function of hidden state. $Unk_P(a)$ denotes a *unique* unknown predicate, which represents the hidden state responsible for the change in truth value of P. It must be unique to avoid biasing correlation of independent unknown effects.

It is clear from the above definition how a **cause** effect may make P unknown. What may not be clear is how a **cause** effect can make P known. In fact, it wouldn't, if not for the fact that the agent knows all the effects of an action (Equation 3.9). However, knowledge of a conditional effect does not necessarily mean knowledge of the consequent. For example, if an agent executes `compress myfile`, it only knows that *if* it had write permission prior to executing `compress`, then `myfile` is compressed afterward.

---

[9] In principle, we could represent all uncertain effects as conditional effects with unknown preconditions, but doing so would be cumbersome. However, we define the *semantics* of uncertain effects in precisely this manner.

<table>
<tr><td>action-schema</td><td>::=</td><td><code>(action</code> name <code>([</code>varlist<code>])</code></td></tr>
<tr><td></td><td></td><td><code>[precond:</code> GD<code>]</code></td></tr>
<tr><td></td><td></td><td><code>effect:</code> effect<code>)</code></td></tr>
<tr><td></td><td></td><td><code>execute:</code> function-symbol<code>(</code>arglist<code>)</code></td></tr>
<tr><td></td><td></td><td><code>sense:</code> {rtvlist} <code>:=</code> function-symbol<code>(</code>arglist<code>)</code></td></tr>
<tr><td>effect</td><td>::=</td><td><code>cause</code> (literal) | <code>observe</code> (literal)</td></tr>
<tr><td>effect</td><td>::=</td><td>effect $\wedge$ effect | (effect) | <code>when</code> (GD) effect</td></tr>
<tr><td>effect</td><td>::=</td><td>$\forall$ (varlist) <code>when</code> (GD) effect | $\exists$ (rtvarlist) effect</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>alit</td><td>::=</td><td><code>satisfy</code> (literal) | <code>initially</code> (literal) |</td></tr>
<tr><td></td><td></td><td><code>hands-off</code> (literal) | literal</td></tr>
<tr><td>GD</td><td>::=</td><td>alit | GD $\wedge$ GD | GD $\vee$ GD | (GD) | $\neg$GD</td></tr>
<tr><td>GD</td><td>::=</td><td>GD $\Rightarrow$ GD | vc $\approx$ vc | vc $\not\approx$ vc</td></tr>
<tr><td>GD</td><td>::=</td><td>$\forall$ (ptvarlist) GD | $\exists$ (ptvarlist) GD</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>literal</td><td>::=</td><td>predicate-symbol <code>([</code>arglist<code>]), [</code>truth-value<code>]</code></td></tr>
<tr><td>truth-value</td><td>::=</td><td><code>T</code> | <code>F</code> | <code>U</code> | var</td></tr>
<tr><td>var</td><td>::=</td><td>ptvar | rtvar</td></tr>
<tr><td>ptvar</td><td>::=</td><td>variable-symbol</td></tr>
<tr><td>rtvar</td><td>::=</td><td>!variable-symbol</td></tr>
<tr><td>varlist</td><td>::=</td><td><code>[</code>type-symbol<code>]</code> var | <code>[</code>type-symbol<code>]</code> var, varlist</td></tr>
<tr><td>rtvlist</td><td>::=</td><td>type-symbol rtvar | type-symbol rtvar, rtvlist</td></tr>
<tr><td>arglist</td><td>::=</td><td>var | constant-symbol | argli.st, arglist</td></tr>
<tr><td>ptvarlist</td><td>::=</td><td>type-symbol (ptvar) | type-symbol (ptvar), gvarlist</td></tr>
</table>

Figure 3.2: EBNF specification of SADL.

3.4 Temporal Projection

We have discussed the function DO, which maps a situation and an action (or sequence of actions) to a new situation, but we haven't yet said how the two situation terms relate to each other. If $s' = \text{DO}(\{a\}_1^n, s)$, we want to answer the following questions.

- Progression: What can we say about s', given knowledge of the conditions that hold in s?

- Regression: What must be true in s to guarantee some desired condition in s'?

We treat each in turn.

3.4.1 Projection & the Frame Problem

The definitions for preconditions and effects that we have given are insufficient to solve the temporal projection problem. SADL effects only list fluents that an action affects, but what about fluents it doesn't affect? Explicitly stating everything that *doesn't* change would be tedious — this is the well-known frame problem. The standard approach to the frame problem, and the one we adopt, is to make the STRIPS assumption: anything not explicitly said to change remains the same. To fully specify the SADL semantics, it is necessary to express the STRIPS assumption in terms of the situation calculus. We use the formulation introduced in [79], and augmented in [80] to account for sensing actions. This strategy consists of providing a formula for each fluent, called a *successor state axiom*, that specifies the value of the fluent in terms of 1) the action executed, and 2) the conditions that held before the action was executed. By quantifying over actions, we can produce a single, concise formula for each fluent that includes only the relevant information. Producing such a formula from the effects of an action depends on a STRIPS assumption (which in the situation calculus formulation is called the Completeness Assumption). That is, it must be the

case that all the updates produced by the action are specified in its effects. However, once we have the successor state axiom, we no longer need the Completeness Assumption, since this assumption is stated explicitly in the "if and only if" form of the successor state axiom.

Specifying update axioms for each fluent independently requires fluents to be logically independent of each other, so disjunction is not allowed. Effects consist of conjunctions of terms, each term being equivalent to one of the following

$$\textbf{when } \gamma_P^{\mathtt{T}}(a) \textbf{ cause}(P, \mathtt{T}) \tag{3.17}$$

$$\textbf{when } \gamma_P^{\mathtt{F}}(a) \textbf{ cause}(P, \mathtt{F}) \tag{3.18}$$

$$\textbf{when } \gamma_P^{\mathtt{U}}(a) \textbf{ cause}(P, \mathtt{U}) \tag{3.19}$$

$$\textbf{when } \kappa_P^{tv}(a) \textbf{ observe}(P, tv) \tag{3.20}$$

where a is an action and P is a fluent, which may contain universally quantified variables or constants,[10] and $\gamma_P^{tv}(a)$ and $\kappa_P^{tv}(a)$ are arbitrary expressions.[11] For example, if `compress tex/*` changes the size of all writable files in directory `tex`, then $\gamma_{\text{size}(f)}^{\mathtt{U}}(\texttt{compress tex/*}) = \text{in.dir}(f, \texttt{tex}) \wedge \text{writable}(f)$. Clearly, all actions can be represented by specifying the γ and κ preconditions for each fluent in the domain theory. If a has a non-conditional effect, **cause**(P, tv), then $\gamma_P^{tv}(a) = \mathtt{T}$. We can express the fact that action a doesn't affect P at all by saying $\forall tv(\gamma_P^{tv}(a) = \mathtt{F})$. We don't list **observe**$(P, \mathtt{T})$ above, since it is subsumed by the conjunction **observe**$(P, v) \wedge v = \mathtt{T}$ (similarly for $\mathtt{F}$). We can assume, without loss of generality, that for any proposition P, there is only one expression of the form **when** $\gamma_P^{\mathtt{T}}(a)$ **cause**$(P, \mathtt{T})$, since if there were more than one, they could easily be combined into one. Combining this fact with the Completeness Assumption, it follows that the **when** clauses are in fact bi-implications. That is, the effect **cause**$(P, \mathtt{T})$ will *only* occur if $\gamma_P^{\mathtt{T}}(a)$ is true.

---

[10] Including variables that will resolve to constants.

[11] with the restriction that effects must be consistent, so, for example, $\gamma_P^{\mathtt{T}}(a) \wedge \gamma_P^{\mathtt{F}}(a)$ must always be false.

Given these definitions, we can state the conditions under which an action changes or preserves a fluent's truth value. Following Pednault [69], we define Σ^a_φ to be the conditions under which an executable action a will establish φ, and Π^a_φ to be the conditions under which a will preserve φ. We have the following establishment conditions:

$$\Sigma^a_\varphi(s) \;\Leftrightarrow\; \gamma^{\mathtt{T}}_\varphi(a,s) \vee (Unk_\varphi(a,s) \wedge \gamma^{\mathtt{U}}_\varphi(a,s)) \quad (3.21)$$

$$\Sigma^a_{\neg\varphi}(s) \;\Leftrightarrow\; \gamma^{\mathtt{F}}_\varphi(a,s) \vee (\neg Unk_\varphi(a,s) \wedge \gamma^{\mathtt{U}}_\varphi(a,s)) \quad (3.22)$$

where $Unk_\varphi(a)$ is the unknown predicate introduced in Equations 3.15 and 3.16. The presence of an effect with a `U` truth value will make φ true or false, depending on the value of $Unk_\varphi(a)$. Since $Unk_\varphi(a)$ is unknown by definition, effects with `U` truth values aren't generally useful for goal establishment. We also have the following preservation conditions:

$$\Pi^a_\varphi(s) \;\Leftrightarrow\; \neg\gamma^{\mathtt{F}}_\varphi(a,s) \wedge (Unk_\varphi(a,s) \vee \neg\gamma^{\mathtt{U}}_\varphi(a,s)) \quad (3.23)$$

$$\Pi^a_{\neg\varphi}(s) \;\Leftrightarrow\; \neg\gamma^{\mathtt{T}}_\varphi(a,s) \wedge (\neg Unk_\varphi(a,s) \vee \neg\gamma^{\mathtt{U}}_\varphi(a,s)) \quad (3.24)$$

As one might expect: $\Pi^a_\varphi \Leftrightarrow \neg\Sigma^a_{\neg\varphi}$.

For each fluent, we can then generate an expression that specifies precisely when it is true or false, by quantifying over actions. For each fluent P, there is a *successor state axiom*, which combines update axioms and frame axioms for P. The successor state axioms are straightforward statements of the STRIPS assumption: a fluent is true if and only if it was made true, or it was true originally and it wasn't made false:

Theorem 3.1 (Successor State Axiom) *Let P be an arbitrary predicate.*

$$\begin{aligned}\mathrm{ACHV}(\pi_a, s, \{\}) \;\Rightarrow\; & [P(DO(a,s)) \Leftrightarrow \Sigma^a_P(s) \vee \\ & P(s) \wedge \Pi^a_P(s)]\end{aligned}$$

Similarly, there is a successor state axiom for K.

Theorem 3.2 (Successor State Axiom for K)

$$\begin{aligned} \text{ACHV}(\pi_a, s, \{\}) \Rightarrow [K(s'', DO(a,s)) \Leftrightarrow \exists s'\, K(s', s) \\ \wedge (s'' = DO(a, s')) \wedge \\ \forall P(\kappa_P^v(a,s) \Rightarrow [P(s') \Leftrightarrow v])] \end{aligned}$$

We have stated this formula in second-order logic, but only because the formula depends on all of the actual fluents in the domain theory. Given any specific domain, this second-order formula could be replaced with an equivalent first-order formula by replacing P with each fluent in the domain.

The situations s'' above are the states consistent with the agent's knowledge after executing a from situation s, so the situations s' are those previous states that the agent considers to have been possible *after* it has executed a. Because the agent's knowledge is correct, one of these situations s' will be s itself. If we substitute DO(a, s) for s'', we find that $s' = s$, which means $P(s) \Leftrightarrow v$. Putting this result back in Equation 3.2 gives us $P(s') \Leftrightarrow P(s)$, meaning that the value that P had in situation s is known to the agent after executing a.

The above definition only specifies when information is gained, and seems to say nothing about when it is lost. However, information loss is indeed accounted for, through the successor state axiom for P. If P becomes true in some situations s' such that K(s', s) (*i.e.*, in some possible worlds), and false in others, then by definition, P is unknown. For example, `compress myfile` compresses `myfile` if it is writable. If it is unknown whether `myfile` is writable, then in some possible worlds, `myfile` is writable and will be compressed. In other worlds, `myfile` is not writable and won't be compressed. The result is that it becomes unknown whether `myfile` is compressed. Similarly, if P was known previously and not changed then, by the successor state axioms for P and K, P will continue to be known. [80].

The above formula correctly describes how K changes, but it is a little unwieldy if what we want to know about is KNOW(φ). Intuitively, KNOW(φ) becomes true

if φ is known to become true or if φ is observed. Additionally, φ continues to be known true until it possibly becomes false. The following formulas follow from the successor state axioms for φ and K.

$$\Pi^a_{\mathrm{KNOW}(\varphi)}(s) \Leftrightarrow \mathrm{KNOW}(\Pi^a_\varphi, s) \tag{3.25}$$

$$\begin{aligned}\Sigma^a_{\mathrm{KNOW}(\varphi)}(s) \Leftrightarrow\ & \mathrm{KNOW}(\gamma^{\mathtt{T}}_\varphi(a), s) \vee \\ & (\kappa^{tv}_{\gamma^{\mathtt{T}}_\varphi(a,s)}(a) \wedge \gamma^{\mathtt{T}}_\varphi(a,s)) \vee (\kappa^{tv}_\varphi(a,s) \wedge \varphi \\ & \wedge \Pi^a_{\mathrm{KNOW}(\varphi)}(s))^{12}\end{aligned} \tag{3.26}$$

The causation precondition for $\mathrm{KNOW}(\varphi)$ is interesting in that it allows the precondition $\gamma^{\mathtt{T}}_\varphi(a)$ to be observed immediately *after* the action is executed, rather than insisting that it be known true beforehand. Actually, there is no reason that the observation need be *immediately* after execution, except that it introduces messy correlations that the agent can't represent. The knowledge representation discussed in Chapter 2 can't represent facts like $\mathrm{KNOW}(P \Leftrightarrow Q)$ when $\neg\mathrm{KNOW}(P)$ and $\neg\mathrm{KNOW}(Q)$, so to keep things simple, we will assume for the time being that no such correlations are introduced by executing actions. It turns out that this condition can be enforced by ensuring that actions aren't executed when the preconditions of their causal effects are unknown, unless those preconditions are verified immediately afterward. Before stating the above more formally, we introduce a few useful definitions.

Definition 3.3 (Plan Fluent) *A plan fluent of* $\{a\}_1^n$ *is any fluent appearing in the preconditions or effects of any action in* $\{a\}_1^n$*.*

---

[12] The additional requirement $\Pi^a_{\mathrm{KNOW}(\varphi)}$ may come as a surprise, since an action that simultaneously observes φ and causes φ to become false or unknown would seem to violate our rule against inconsistent actions. However, such effects aren't inconsistent, since the observation pertains to situation s, whereas the update is to situation $\mathrm{DO}(a, s)$. Such *destructive sensing actions* are commonplace. For example, biologists find out the number of insects living in a tree by fogging the tree with poison and counting the insects that fall out.

Definition 3.4 (Correlation) *Two fluents P and Q are said to be correlated in situation s if* $\mathrm{KNOW}(P \Rightarrow Q, s)$ *or* $\mathrm{KNOW}(Q \Rightarrow P, s)$. *$P$ and Q are correlated unknown fluents if they are correlated but their truth values are unknown in situation s.*

Theorem 3.5 (No Correlations) *Assume the following conditions hold: There are no correlations between unknown plan fluents in situation s_0, no disjunctive effects, and no action a_i in $\{a\}_1^n$(or any effect of a_i) has any preconditions that are unknown in the situation $DO(a_i, s_i)$. Then there will be no correlations between unknown plan fluents at any time during the execution of $\{a\}_1^n$.*

An immediate consequence of the lack of correlations is that $\mathrm{KNOW}(A \vee B) \Rightarrow \mathrm{KNOW}(A) \vee \mathrm{KNOW}(B)$. Since $\mathrm{KNOW}(\neg A \Rightarrow B)$, the truth value of A or of B must be known true or false. It follows immediately that one must be known true.

are correlated, they cannot both be unknown

As we will see in Chapter 5, this restriction can be relaxed, since the structures used by the planner provide a limited ability to keep track of these correlations, but for now we will assume that no correlations exist. Given this assumption, we provide the following successor state axiom for KNOW.

Theorem 3.6 (Successor State Axiom for KNOW) *If there are no correlations between unknown plan fluents in the agent's knowledge, then*

$$\begin{aligned}\mathrm{ACHV}(\pi_a, s, \{\}) \quad \Rightarrow \quad & [\mathrm{KNOW}(P, DO(a, s)) \Leftrightarrow \Sigma^a_{\mathrm{KNOW}(P)}(s) \vee \\ & \mathrm{KNOW}(P, s) \wedge \Pi^a_{\mathrm{KNOW}(P)}(s)]\end{aligned}$$

If there did happen to be correlations in the agent's knowledge, then the "$\Leftrightarrow$" above would be replaced with a "$\Leftarrow$" — that is, the condition above is sufficient, but not necessary, for knowing P.

3.4.2 Regression

Most modern planners build plans using goal regression — starting with a goal and successively adding actions that achieve either part of the goal or preconditions of previously added actions. Once all preconditions are satisfied by some action (or were true in the initial state) and preserved by all intervening actions, the plan is complete. It is therefore useful to have a formal specification of what conditions must be true for a given action sequence to achieve a given goal. Let R_a be a *regression operator* for action a. $\mathsf{R}_a(\varphi)$ is a formula that, if true immediately before the execution of a, results in φ being true after a is executed. We define $\mathsf{R}_{\{a\}_1^n}(\varphi)$ to be $\mathsf{R}_{a_1}(\mathsf{R}_{a_2}(\ldots(\mathsf{R}_{a_n}(\varphi))))$. Naturally, regression on an action sequence of zero length is the identity function: $\mathsf{R}_{\{\}}(\varphi) = \varphi$.

Let α be an axiomatization of the initial conditions, and let Γ be some goal expression. The objective of planning is to produce an executable sequence of actions, $\{a\}_1^n$, such that $\alpha \models \mathsf{R}_{\{a\}_1^n}(\Gamma)$. We discuss executability in Section 3.4.3.

We specify regression operators for **satisfy**, **initially** and **hands-off** goals below. Since some conditions could be true in the initial state, we also must specify when a condition is true after executing a plan of zero length. Since **initially** indicates something that must be true before the plan is executed, and **satisfy** indicates things true afterwards, it follows that if there are no actions in the plan, then **initially** and **satisfy** have the same interpretation: For all φ,

$$\mathsf{R}_{\{\}}(\textbf{initially}(\varphi)) = \text{KNOW}(\varphi) \tag{3.27}$$

$$\mathsf{R}_{\{\}}(\textbf{satisfy}(\varphi)) = \text{KNOW}(\varphi) \tag{3.28}$$

hands-off is always true in the initial state, since it can only be violated by changing the proscribed fluent:

$$\mathsf{R}_{\{\}}(\textbf{hands-off}(\varphi)) = \mathtt{T} \tag{3.29}$$

We now consider how to regress a SADL goal formula through an action. A goal

satisfy(φ) is achieved if the agent knows that φ is true; *i.e.*, φ just became true, was just observed to be true, or was previously known to be true and wasn't subsequently affected. The first two conditions are captured by $\Sigma^a_{\text{KNOW}(\varphi)}$. The latter holds when **satisfy**(φ) held in the previous state, and knowledge of φ was preserved:

$$\begin{aligned} \mathsf{R}_a(\textbf{satisfy}(\varphi)) &= \Sigma^a_{\text{KNOW}(\varphi)} \vee (\textbf{satisfy}(\varphi) \\ &\quad \wedge \Pi^a_{\text{KNOW}(\varphi)}) \end{aligned} \tag{3.30}$$

A **hands-off** goal holds if the truth value of φ always remains the same as it was in the initial state. **hands-off**(φ) doesn't forbid actions that *affect* φ — just actions that *change* φ. For example, an action `compress myfile` doesn't violate the goal **hands-off**(compressed(`myfile`)) if `myfile` was already compressed initially.[13]

$$\begin{aligned} \mathsf{R}_a(\textbf{hands-off}(\varphi)) &= (\text{KNOW}(\Pi^a_{\neg\varphi}) \vee \textbf{initially}(\varphi)) \\ &\quad \wedge(\text{KNOW}(\Pi^a_{\varphi}) \vee \textbf{initially}(\neg\varphi)) \\ &\quad \wedge \textbf{hands-off}(\varphi) \end{aligned} \tag{3.31}$$

initially(φ) is satisfied after the execution of action a if it was already satisfied, or if φ was observed by action a, and wasn't affected by any previous actions. Unlike other goals, we are interested in the *first* time point at which an **initially** goal is achieved, as opposed to the last. The disjunct **initially**(φ) ensures that the first occurrence is considered, because it is always regressed back.

$$\begin{aligned} \mathsf{R}_a(\textbf{initially}(\varphi)) &= \textbf{initially}(\varphi) \vee (\kappa^{tv}_{\varphi}(a) \wedge \varphi \\ &\quad \wedge \textbf{hands-off}(\varphi)) \end{aligned} \tag{3.32}$$

This definition doesn't rule out using destructive sensing actions. All that matters is that φ be undisturbed *before* it is sensed. It's fine if the act of sensing the value of φ itself affects φ.

[13] This is a departure from UWL's notion of **hands-off**, in which the `compress` would be a violation. However, uncompressing the file and then recompressing it does violate the goal, since the uncompress changes the fluent.

Unannotated preconditions merely need to be satisfied in the final state, and it isn't necessary that they be known true.

$$\mathsf{R}_a(\varphi) \quad = \quad \Sigma^a_\varphi \vee (\varphi \wedge \Pi^a_\varphi) \tag{3.33}$$

Logical operators are simply regressed back to the initial state, since their interpretation is the same across all situations, as detailed in [70].

With these definitions, we can show that regression is correct — that is, if the formula returned by $\mathsf{R}_a(\Gamma)$ is true, and $\{a\}_1^n$ is successfully executed, then Γ will indeed be true.

Theorem 3.7 (Soundness of Regression) *Let $\{a\}_1^n$ be an executable action sequence. Let Γ be a goal formula, and let α be an axiomatization of the domain, including the successor state axioms. Then*

$$\alpha \models (\text{ACHV}(\mathsf{R}_{\{a\}_1^n}(\Gamma), s_0, \{\}) \Rightarrow \text{ACHV}(\Gamma, s_0, \{a\}_1^n))$$

We would like the reverse to be true as well — *i.e.*, if Γ is true after $\{a\}_1^n$ is executed, then $\mathsf{R}_a(\Gamma)$ must have been true. However, as with the successor state axiom for KNOW, that is not the case unless we make the additional restriction that no actions can be executed when doing so would introduce correlations between unknown fluents (see Theorem 3.5), since that would require more sophisticated reasoning to ensure completeness of regression.

Theorem 3.8 (Completeness of Regression) *Let $\{a\}_1^n$ be an executable action sequence. Let Γ be a goal formula, and let α be an axiomatization of the domain, including the successor state axioms. If the conditions specified in Theorem 3.5 hold, then $\alpha \models (\text{ACHV}(\Gamma, s_0, \{a\}_1^n) \Rightarrow \text{ACHV}(\mathsf{R}_{\{a\}_1^n}(\Gamma), s_0, \{\}))$*

3.4.3 Executability

Regression operators alone only tell part of the story about when an action, or sequence of actions, can achieve a goal. $\mathsf{R}_a(\varphi)$ consists of the conditions under which a will achieve φ, *assuming it is successfully executed.* So to ensure that a brings about φ, we must also ensure that a can be executed. Action a is executable in situation s iff the preconditions of a, π_a, are true in s. A sequence of actions, $\{a\}_1^n$, is executable in s iff a_1 is executable in s, a_2 is executable in $\text{DO}(a_1, s)$, a_3 is executable in $\text{DO}(a_2, \text{DO}(a_1, s))$, and so on.

3.5 LCW *Updates*

As the agent is informed of the changes to the external world — through its own actions or through the actions of other agents — it can gain and lose information about the world, and will update its database $\mathcal{M}$ of ground literals (see Chapter 2). For example, after executing the UNIX command `finger weld@june`, the agent should update $\mathcal{M}$ with the newly observed truth value for `active.on(weld, june)`. Similarly, an agent's actions can cause it to gain or lose LCW. When a file is compressed, for example, the agent loses information about the file's size; when all postscript files are deleted from a directory, the agent gains the information that the directory contains no such files.

This section presents a sound and efficient method for updating $\mathcal{L}$, the agent's store of LCW sentences. In Section 3.5.1, we start by showing how complex updates can be partitioned into atomic components. The next four subsections (3.5.2–3.5.4) present policies for handling the four different types of atomic updates (see Table 3.1 for a summary). Section 3.5.6 provides an example illustrating the update mechanism. Then, in Section 3.5.7, we show that the updates can be performed in polynomial time. The final two sections discuss the optimality of our policies: Section 3.5.8 demonstrates that no valid LCW sentences are discarded by the atomic update policies,

and Section 3.5.9 presents an optimal order for handling the atomic components of a complex update.

3.5.1 Representation of Change

The LCW knowledge must be updated based on changes resulting from executing SADL actions. We show that these changes can be decomposed into a set of atomic updates, each concerning the truth value of a set of literals matching a pattern. For example, suppose that initially the agent doesn't know whether `weld` is active on the machine called `june`, so it executes a UNIX `finger` action, which observes that `weld` *is* active. We can describe the resulting change in the agent's information with a single atomic update: $\Delta(\texttt{active.on(weld, june)}, \texttt{U} \rightarrow \texttt{T})$, meaning that the truth value of the proposition `active.on(weld, june)` changed from `U` to `T`.

Below, we define this Δ notation formally, but before delving into the technical details, note that the description of more complex changes may require multiple atomic components. For example, consider the UNIX action `mv /papers/kr94.tex /archive/kr94.tex`, which has the effect of moving a file from one directory to another. The change due to the execution of this action can't be represented as a single update by our definitions, but it can be expressed as the following set of atomic updates:

- $\Delta(\texttt{in.dir(kr94.tex, /papers)}, \texttt{T} \rightarrow \texttt{F})$
- $\Delta(\texttt{in.dir(kr94.tex, /archive)}, \texttt{F} \rightarrow \texttt{T})$
- $\Delta(\texttt{in.dir(kr94.tex, /archive)}, \texttt{U} \rightarrow \texttt{T})$

The last two atomic components capture the fact that, regardless of whether the agent knew whether a file named `kr94.tex` was present in `/archive` before the `mv`, the agent knows that such a file is present afterward. Informally, Δ(`in.dir(kr94.tex,`

`/archive)`,F → T) should be read as "If the agent knew (before the `mv`) that no file named `kr94.tex` was in `/archive`, then the agent now knows (after the `mv`) that there is such a file in `/archive`."

As mentioned in Section 3.4.1, we require that the atomic updates corresponding to a compound change are consistent, *i.e.*, at most one atomic update changes the truth value of any single ground formula. Given this assumption, our update policy is free to process the atomic components in any order.[14] Furthermore, we assume that these atomic updates constitute a complete list of changes in the world, thus sidestepping the ramification problem [33].[15] In the example above, each of the three atomic updates changed the truth value of at most one ground literal, but in general an atomic update need not be ground; in other words, a single atomic update can affect the truth value of an unbounded number of ground literals. For example, suppose that $\texttt{size}(\texttt{paper.tex}, 14713) \in \mathcal{S}$ before the agent executes the UNIX command `compress paper.tex`. In this case, numerous literals change their truth value when the size of `paper.tex` becomes unknown: `size(paper.tex, 14713)` changes from T to U, while `size(paper.tex, 14712)` (and an unbounded number of similar literals) change from F to U. In this case, we summarize the change with the following pair of updates, the last of which affects the truth value of an infinite number of ground literals:

- $\Delta(\texttt{size(paper.tex, }\ x\texttt{)}, \texttt{T} \rightarrow \texttt{U})$
- $\Delta(\texttt{size(paper.tex, }\ x\texttt{)}, \texttt{F} \rightarrow \texttt{U})$

So far our discussion of atomic updates has been informal, but we now make the

---

[14] Section 3.5.9 explains how transformations exploiting this commutativity can lead to improved performance.

[15] This is standard in the planning literature. For example, a STRIPS operator that moves block A from B to C must delete `on(A, B)` and also add `clear(B)` even though `clear(B)` can be defined as $\forall y\ \neg\texttt{on(}y\texttt{,B)}$.

notion precise.[16] We model a change from an agent's incomplete theory, $\mathcal{S}$, to a new theory, $\mathcal{S}'$, as follows. Let φ be a positive literal possibly containing free variables, for example, $\varphi =$`size(paper.tex,` x`)`. We define the sets $\mathcal{T}(\varphi,\, \mathcal{S})$, $\mathcal{F}(\varphi,\, \mathcal{S})$, and $\mathcal{U}(\varphi,\, \mathcal{S})$ as the ground instances of φ that are true, false or unknown, respectively:

$$
\begin{aligned}
\mathcal{U}(\varphi, \mathcal{S}) &\equiv \{\psi | \psi = \varphi\theta \land \psi \notin \mathcal{S} \land \neg\psi \notin \mathcal{S}\} \\
\mathcal{T}(\varphi, \mathcal{S}) &\equiv \{\psi | \psi = \varphi\theta \land \psi \in \mathcal{S}\} \\
\mathcal{F}(\varphi, \mathcal{S}) &\equiv \{\psi | \psi = \varphi\theta \land \neg\psi \in \mathcal{S}\}
\end{aligned}
$$

Note that for any value of x, `size(paper.tex,` x`)` will be in exactly one of the three sets. Finally, $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{U})$ means that all elements of $\mathcal{F}(\varphi,\, \mathcal{S})$ are elements of $\mathcal{U}(\varphi,\, \mathcal{S}')$, *i.e.*, all matches to φ that are false before execution become unknown afterwards.

To define Δ precisely, we need one more notational device. For convenience in representing those literals that remain unchanged from $\mathcal{S}$ to $\mathcal{S}'$, we define the operator $\ominus$ which, given a theory $\mathcal{S}$ and a set of positive, ground literals $\mathcal{N}$, returns $\mathcal{S}$ with all positive *and negated* members of $\mathcal{N}$ removed:

$$
\mathcal{S} \ominus \mathcal{N} = \{\psi | \psi \in \mathcal{S} \land \psi \notin \mathcal{N} \land \neg\psi \notin \mathcal{N}\} \tag{3.34}
$$

To understand the intuition behind $\ominus$, consider the previous example in which $\varphi =$`size(paper.tex,` x`)`. $\mathcal{T}(\varphi,\mathcal{S}) = \{$`size(paper.tex, 14713)`$\}$, so $\mathcal{S} \ominus \mathcal{T}(\varphi, \mathcal{S})$ is equivalent to $\mathcal{S}$ with the information `size(paper.tex, 14713)` removed. Thus $\mathcal{S}' \ominus \mathcal{T}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{T}(\varphi, \mathcal{S})$ is simply a concise way of saying that the only change from $\mathcal{S}$ to $\mathcal{S}'$ concerns the belief that `paper.tex` has the size 14713. Neither the belief that `paper.tex` is in directory `/papers`, nor the belief that `paper.tex` *doesn't*

[16] Readers satisfied with this informal explanation may wish to skip to Section 3.5.2.

have size 14712 has changed. In other words, $\mathcal{S}' \ominus \mathcal{T}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{T}(\varphi, \mathcal{S})$ is a frame axiom stating that nothing has changed aside from the truth value of literals contained in $\mathcal{T}(\varphi, \mathcal{S})$.

The formal definition of $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{F})$ appears below.

$$\begin{aligned} \Delta(\varphi, \mathtt{T} \rightarrow \mathtt{F}) \equiv\ & \mathcal{T}(\varphi, \mathcal{S}') = \{\} \wedge \\ & \mathcal{F}(\varphi, \mathcal{S}') = \mathcal{F}(\varphi, \mathcal{S}) \cup \mathcal{T}(\varphi, \mathcal{S}) \wedge \\ & \mathcal{S}' \ominus \mathcal{T}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{T}(\varphi, \mathcal{S}) \end{aligned} \tag{3.35}$$

Definitions for most other truth values are similar, but one bears discussion: $\Delta(\varphi, \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$. There is no need to specify the change *from* a disjunction of truth values because such a change can be decomposed into a pair of simpler updates. Specifically, there is no need to define $\Delta(\varphi, (\mathtt{T} \vee \mathtt{F}) \rightarrow \mathtt{U})$ because it would be equivalent to the set containing both $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{U})$ and $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{U})$. However, some useful changes cannot be modeled without using a disjunction on the right hand side of the arrow. For example, the UNIX `ls -a` command observes the name of all files in a directory argument; when applied to the `/tex` directory, the command induces the update $\Delta(\mathtt{in.dir}(o, \mathtt{/tex}), \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$, because some files are observed to be present while all others are now known to be absent. We define the update formally as follows:

$$\begin{aligned} \Delta(\varphi, \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F})) \equiv\ & \mathcal{U}(\varphi, \mathcal{S}') = \{\} \wedge \\ & \mathcal{T}(\varphi, \mathcal{S}') \supseteq \mathcal{T}(\varphi, \mathcal{S}) \wedge \\ & \mathcal{F}(\varphi, \mathcal{S}') \supseteq \mathcal{F}(\varphi, \mathcal{S}) \wedge \\ & \mathcal{S}' \ominus \mathcal{U}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{U}(\varphi, \mathcal{S}) \end{aligned} \tag{3.36}$$

In subsequent sections, we describe a process for handling these updates. Specifically, we assume that the agent starts with a $\mathcal{M}$, $\mathcal{L}$ pair that forms a conservative

representation of an incomplete theory $\mathcal{S}$. When informed of a change, *i.e.* a set of atomic updates described using the Δ notation defined above, the agent must create a new $\mathcal{M}'$ and $\mathcal{L}'$, ensuring that these databases are still conservative representations, yet retain as much information as possible. We present our method for processing updates as a set of rules and state them as theorems since they are sound: *i.e.*, they preserve conservative representations.

By distinguishing between transitions to and from $\mathtt{U}$ truth values, $\mathcal{L}$ updates can be divided into four mutually exclusive and exhaustive cases, which we call Information Gain, Information Loss, Domain Growth, and Domain Contraction:

- Information Gain: $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$.

- Information Gain: $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{U})$ or $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{U})$.

- Domain Growth: $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{T})$.

- Domain Contraction: $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{F})$.

These update classes are easily generated from SADL actions, as the following theorems attest

Theorem 3.9 (Updates generated by cause(φ, $\mathtt{T}$)) *If the only effect of an executed action is of the form* **cause**$(\varphi, \mathtt{T})$*, then updates to* $\mathcal{M}$ *will be of the form* $\Delta(\varphi, \mathtt{U} \vee \mathtt{F} \rightarrow \mathtt{T})$*, which can be decomposed into a combination of Domain Growth and Information Gain updates.*

Theorem 3.10 (Updates generated by cause(φ, $\mathtt{F}$)) *If the only effect of an executed action is of the form* **cause**$(\varphi, \mathtt{F})$*, then updates to* $\mathcal{M}$ *will be of the form* $\Delta(\varphi, \mathtt{U} \vee \mathtt{T} \rightarrow \mathtt{F})$*, which can be decomposed into a combination of Domain Contraction and Information Gain updates.*

Theorem 3.11 (Updates generated by cause(φ, U)) *If the only effect of an executed action is of the form* **cause**(φ, U)*, then all updates to* $\mathcal{M}$ *will be of the form* $\Delta(\varphi, \mathtt{T} \vee \mathtt{F} \rightarrow \mathtt{U})$ *(Information Loss)*

Theorem 3.12 (Updates generated by observe(φ, tv)) *If the only effect of an executed action is of the form* **observe**(φ, tv)*, then updates to* $\mathcal{M}$ *will be of the form* $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$ *(Information Gain).*

In the next four sections, we consider each case in turn.

3.5.2 Information Gain

An agent gains information when it executes an information-gathering action (*e.g.*, UNIX `wc` or `ls`), or when a change to the world results in information gain. In general, if an agent gains information, it cannot lose LCW, and will gain LCW in some cases as explained below.

Theorem 3.13 (Information Gain Rule) *Let* $\mathcal{L}$ *be part of a conservative representation. If an atomic change is of the form* $\Delta(\varphi, \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$*, then* $\mathcal{L}' \leftarrow \mathcal{L} \cup \{\text{LCW}(\varphi)\}$ *yields a conservative representation.*

The Information Gain Rule is obviously true when φ is ground, in which case this LCW update would be vacuous. However, the rule can also apply when φ contains universally quantified variables. For example, execution of the UNIX command `ls -a /tex` produces a $\Delta(\mathtt{in.dir}(f, \mathtt{/tex}), \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$ update, where f is a universally quantified variable. As a result, the Information Gain Rule concludes that the agent now knows all files in the `/tex` directory: LCW($\mathtt{in.dir}(f, \mathtt{/tex})$).

If the unique value of a function is determined, such as the word count of a file, then a ground update can lead to LCW of a lifted sentence. For example, if an agent discovers that `foo.tex` has length `5512` then it knows that the length is neither `5513` nor any other value. In other words, the agent knows LCW(`word.count(foo.tex,` x`)`).

In order to make this precise, we define the cardinality of a (lifted) literal, φ, in a set of sentences (*e.g.*, $\mathcal{M}$ or $\mathcal{W}$) to be the number of ground literals in the set that unify with φ.

$$\texttt{Cardinality}(\varphi, \mathcal{M}) = |\{\phi \in \mathcal{M} \text{ such that } \phi \text{ is ground and } \exists\theta\ \phi = \varphi\theta\}|$$

When an update causes the cardinality of φ to be the same in $\mathcal{M}$ as it is in $\mathcal{W}$, we can conclude that we have $\text{LCW}(\varphi)$:

Theorem 3.14 (Counting Rule (after [83])) *Let $\mathcal{M}$, $\mathcal{L}$ be a conservative representation and let θ be a substitution. If an atomic change of the form $\Delta(\varphi\theta, \texttt{U} \rightarrow \texttt{T})$ causes $\texttt{Cardinality}(\varphi, \mathcal{M}') = \texttt{Cardinality}(\varphi, \mathcal{W})$ then $\mathcal{L}' \leftarrow \mathcal{L} \cup \{\text{LCW}(\varphi)\}$ yields a conservative representation.*

To utilize the Counting Rule in practice, our agent relies on a set of explicit axioms that define the cardinality of predicates in $\mathcal{W}$, as mentioned in Section 3.2.5. For example, we tell our agent that `word.count` is functional in its first argument, as is `file.size` *etc.*

In some cases the Information Gain and Counting Rules, used in conjunction with the Composition Rule, can lead to additional forms of local closed world information. For example, the UNIX command `ls -la /tex` detects the size of all files in the directory `/tex`. The update $\Delta(\texttt{in.dir}(f, \texttt{/tex}), \texttt{U} \rightarrow (\texttt{T} \vee \texttt{F}))$ allows the Information Gain Rule to conclude $\text{LCW}(\texttt{in.dir}(\text{f}, \texttt{/tex}))$, as explained above. Suppose that there are two files in the directory, `foo.tex` and `bar.tex`, which are `55` and `66` bytes long respectively. The following two updates $\Delta(\texttt{size(foo.tex, 55)}, \texttt{U} \rightarrow \texttt{T})$ and $\Delta(\texttt{size(bar.tex, 66)}, \texttt{U} \rightarrow \texttt{T})$ will yield $\text{LCW}(\texttt{size(foo.tex, } l))$ and $\text{LCW}(\texttt{size (bar.tex, } l))$ via the Counting Rule. The Composition Rule can now be used to conclude

$$\text{LCW}(\texttt{in.dir}(f, \texttt{bin}) \wedge \texttt{size}(f, l))$$

Cases like this (*i.e.*, where LCW results from the execution of an action) are so common that we apply the Composition Rule proactively. In other words, we add the LCW statement above at the time that the Δ updates are received rather than waiting for an LCW query. The procedure for doing this optimization is straightforward:

- If any non-universally quantified variables are in predicate arguments with cardinality 1, then those variables can be replaced with universally quantified variables, provided all other arguments are held constant. This follows from the counting rule. If there is only one value to observe, and that value is in fact observed, then all values are observed.

- If any effect ϕ contains only universally quantified variables or constants, and ϕ has no precondition, then we can add an effect of the form LCW(ϕ). This is a straightforward application of the definition of LCW.

- If ϕ does have a precondition Φ, but LCW(Φ) follows from some effect of the same action, then we can add an effect LCW($\Phi \wedge \phi$). This follows from the composition theorem.

- If ϕ has precondition Φ, but LCW(Φ) does not follow from any effect of the same action, then we can add an effect **when**(LCW(Φ) LCW($\Phi \wedge \phi$).

- If there is any effect of the form **when**(Φ) **observe**(Φ) or **when**(Φ) **cause**(Φ, F) then we can add an effect of the form LCW(Φ).

It is important to note that the policy does not add LCW sentences of arbitrary length to $\mathcal{L}$. The maximum length of an LCW formula generated from an effect is one plus the maximum number of conjuncts appearing in the effect precondition. For example, in the softbot domain tested in Section 7.2, all LCW sentences added to $\mathcal{L}$ had fewer than 3 conjuncts.

3.5.3 Information Loss

An agent loses information when a literal, previously known to be true (or false), is asserted to be unknown. When a UNIX file is compressed, for example, information about its size is lost. In general, when information is lost about some literal, all LCW statements "relevant" to that literal are lost. To make our notion of relevance precise, we begin by defining the set $\texttt{PREL}(\varphi)$ to denote the LCW assertions *potentially relevant* to a positive literal φ:[17]

$$\texttt{PREL}(\varphi) \equiv \{\Phi \in \mathcal{L} \ \mid \exists x \in \Phi, \exists\theta, \exists\alpha, x\theta = \varphi\alpha\}$$

For example, if an agent has complete information on the size of all files in /kr94, and a file lcw.tex in /kr94 is compressed ($\varphi = \texttt{size}(\texttt{lcw.tex}, n)$), then the sentence

$$\text{LCW}(\texttt{in.dir}(f, \texttt{/kr94}) \wedge \texttt{size}(f, c)) \tag{3.37}$$

is in $\texttt{PREL}(\varphi)$ and should be removed from $\mathcal{L}$. Unfortunately, when a file in the directory /bin is compressed, the above LCW sentence is still in $\texttt{PREL}(\varphi)$ ($x = \texttt{size}(f, c)$) even though the agent retains complete information about the files in /kr94. Clearly, LCW sentence 3.37 ought to remain in $\mathcal{L}$ in this case. To achieve this behavior, we check whether the agent has information indicating that the LCW sentence does not "match" the compressed file. If so, the LCW sentence remains in $\mathcal{L}$. In general, we define the set of LCW assertions *relevant* to a positive literal φ to be the following subset of $\texttt{PREL}(\varphi)$:

$$\texttt{REL}(\varphi) \equiv \{\Phi \in \mathcal{L} \ \mid \exists x \in \Phi, \exists\theta, \exists\alpha, x\theta = \varphi\alpha \wedge \mathcal{L} \wedge \mathcal{M} \ \not\models \neg(\Phi - x)\theta\}$$

If θ is not a complete mapping then, to exclude Φ from $\texttt{REL}(\varphi)$, it is necessary that all possible ground instances of $(\Phi - x)\theta$ are known to be false, or equivalently, that

[17] Since the sentences in $\mathcal{L}$ are conjunctions of positive literals, we use the notation $\varphi \in \Phi$ to signify that φ is one of Φ's conjuncts, and the notation $\Phi - \varphi$ to denote the conjunction Φ with φ omitted.

LCW$((\Phi - x)\theta)$, and there is no match to $(\Phi - x)\theta$ in $\mathcal{M}$. We can now state our update policy for Information Loss:

Theorem 3.15 (Information Loss Rule) *Let $\mathcal{L}$ be part of a conservative representation. If an atomic change is either of the form $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{U})$ or $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{U})$, then $\mathcal{L}' \leftarrow \mathcal{L} - \mathtt{REL}(\varphi)$ yields a conservative representation.*

Note that compressing a file `foo` in `/bin` does not remove LCW sentence 3.37. To see this, let $x = \mathtt{size}(f, c)$, $\theta = (\mathtt{foo}/f)$, and $\phi_i = \mathtt{in.dir}(f, \mathtt{/kr94})$. Since `foo` is known to be in `/bin` (and `in.dir` is a functional relation), from $\mathcal{L} \wedge \mathcal{M}$ one can prove that $\neg\phi_i\theta$. Hence, $(\mathcal{L} \wedge \mathcal{M} \not\vdash \neg\phi_i\theta)$ is false and Φ is not included in $\mathtt{REL}(\varphi)$. Note also that, given our assumptions (correct information, *etc.*), information is only lost when the world's state changes.

3.5.4 Changes in Domain

Finally, we have the most subtle cases: an agent's theory changes without strictly losing or gaining information. For example, when the file `ai.sty` is moved from the `/tex` directory to `/kr94`, we have that the updated $\mathcal{M}' \neq \mathcal{M}$ but neither database is a superset of the other. When the theory changes in this way, the domain of sentences containing $\mathtt{in.dir}(f, \mathtt{/kr94})$ grows whereas the domain of sentences containing $\mathtt{in.dir}(f, \mathtt{/tex})$ contracts. LCW information may be lost in sentences whose domain grew. Suppose that, prior to the file move, the agent knows the word counts of all the files in `/kr94`; if it does not know the word count of `ai.sty`, then that LCW assertion is no longer true. As with Information Loss, we could update $\mathcal{L}$ by removing the set $\mathtt{REL}(\varphi)$. However, this policy is overly conservative. Suppose, in the file move described above, that the agent *does* know the word count of `ai.sty`. In this case, it retains complete information over the word counts of the files in `/kr94`, even after `ai.sty` is moved.

More generally, when the domain of an LCW sentence grows, but the agent has LCW on the new element of the domain, the LCW sentence can be retained. To make

this intuition precise, we define the following "minimal" subset of $\mathtt{REL}(\varphi)$:

$$\mathtt{MREL}(\varphi) = \{\Phi \in \mathcal{L} \mid \exists x \in \Phi, \exists \theta, \exists \alpha, x\theta = \varphi\alpha \wedge \mathcal{L} \wedge \mathcal{M} \not\models \textsc{lcw}((\Phi - x)\theta)\}$$

We can now state our update policy for Domain Growth:

Theorem 3.16 (Domain Growth Rule) *Let $\mathcal{L}$ be part of a conservative representation. If an atomic change is of the form $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{T})$, then $\mathcal{L}' \leftarrow \mathcal{L} - \mathtt{MREL}(\varphi)$ yields a conservative representation.*

When the domain of a sentence contracts, no LCW information is lost. For instance, when a file is removed from the directory `/kr94`, we will still know the size of each file in that directory.

Theorem 3.17 (Domain Contraction Rule) *Let $\mathcal{L}$ be part of a conservative representation. If an atomic change is of the form $\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{F})$, then $\mathcal{L}' \leftarrow \mathcal{L}$ yields a conservative representation.*

It might seem that the Domain Contraction Rule misses some important opportunities to gain LCW. Suppose the agent executes `rm /kr94/*`. It ought to realize that it is now familiar with all the files in `/kr94`, because the directory is empty. In fact, the agent *does* realize this because the `rm /kr94/*` command is processed as two separate sets of updates:

- $\Delta(\mathtt{in.dir}(f,\ \mathtt{/kr94}), \mathtt{T} \rightarrow \mathtt{F})$

- $\Delta(\mathtt{in.dir}(f,\ \mathtt{/kr94}), \mathtt{U} \rightarrow \mathtt{F})$

The second update invokes the Information Gain Rule which results in: LCW(in.dir(f, /kr94)).

In summary, the above rules guarantee that $\mathcal{L}$ does not contain invalid LCW assertions, so long as the agent is apprised of any changes to the world state. However, for the sake of tractability, the rules are conservative — $\mathcal{L}'$ may be incomplete. For

Table 3.1: A summary of the mutually exclusive and exhaustive atomic update rules for the LCW database $\mathcal{L}$.

$\Delta(\varphi, \rightarrow)$	Update Rule	$\mathcal{L} \rightarrow \mathcal{L}'$	UNIX Examples
$\mathtt{U} \rightarrow (\mathtt{F} \vee \mathtt{T})$	Information gain	$\mathcal{L}' \leftarrow \mathcal{L} \cup \text{LCW}(\varphi)$	`ls`, `wc`
$(\mathtt{F} \vee \mathtt{T}) \rightarrow \mathtt{U}$	Information loss	$\mathcal{L}' \leftarrow \mathcal{L} - \mathtt{REL}(\varphi)$	`compress`
$\mathtt{T} \rightarrow \mathtt{F}$	Domain contraction	$\mathcal{L}' = \mathcal{L}$	`rm`
$\mathtt{F} \rightarrow \mathtt{T}$	Domain growth	$\mathcal{L}' \leftarrow \mathcal{L} - \mathtt{MREL}(\varphi)$	`cp`

example, if `ai.sty` is moved into `/kr94`, but the word count of `ai.sty` is unknown, we might wish to say that we know the word counts of all the files in `/kr94` *except* `ai.sty`. However, we refrain from storing negated sentences in $\mathcal{L}$ for the sake of speedy LCW inference, as discussed in Section 2.3.

3.5.5 *Initial Closed World Knowledge*

As we describe in Chapter 5, the agent maintains a record of its knowledge about the initial situation. This record includes separate LCW knowledge over the initial state, designated with formulas such as LCW(**initially**(φ)). This knowledge is updated in the same manner that the current LCW knowledge is updated, except that in many ways the updates are simpler. No updates can can retract LCW, since **initially** knowledge never goes away (Lemma 6.3). Any time LCW(Φ) is added to $\mathcal{L}$, it will also be added to $\mathcal{I}$, provided none of the ground literals matching Φ have been changed prior to being observed.

3.5.6 *Example*

Table 3.1 provides a capsule summary of our LCW update rules discussed in the previous four sections. Below, we provide an extended example of the update machinery in action. Specifically, we illustrate how the update rules affect $\mathcal{L}$ as the following

command sequence is executed:

```
ls -al /kr94
ls -al /papers
mv /kr94 /kr.ps /papers
compress /papers /kr.ps
```

Initially, both the databases representing ground formulas ($\mathcal{M}$) and LCW formulas ($\mathcal{L}$) are empty. The execution of `ls -al` in the directory `/kr94` reveals the files in the directory and their size in bytes. For brevity, we will ignore other effects. Suppose that the files are `kr.tex` and `kr.ps`, and their sizes are 100 and 300 respectively. In this case, $\mathcal{M}$ is updated as follows:

$$\begin{aligned}\mathcal{M} = \{&\texttt{in.dir}(\texttt{kr.tex}, \texttt{/kr94}),\\ &\texttt{size}(\texttt{kr.tex}, 100),\\ &\texttt{in.dir}(\texttt{kr.ps}, \texttt{/kr94}),\\ &\texttt{size}(\texttt{kr.ps}, 300)\}\end{aligned}$$

The agent knows the contents of `/kr94`, and the sizes of all the files therein. In addition, because the parent directory and size of each file are unique, the Counting Rule implies that the agent has LCW on the size and parent directory of each file. This information is recorded in the LCW database as follows:

$$\begin{aligned}\mathcal{L} = \{&\textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94})),\\ &\textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94}) \wedge \texttt{size}(f, l))\\ &\textsc{lcw}(\texttt{in.dir}(\texttt{kr.tex}, d)),\\ &\textsc{lcw}(\texttt{size}(\texttt{kr.tex}, l)),\\ &\textsc{lcw}(\texttt{in.dir}(\texttt{kr.ps}, d)),\\ &\textsc{lcw}(\texttt{size}(\texttt{kr.ps}, l))\}\end{aligned}$$

The directory `/papers` is initially empty. Thus, after executing `ls -al` in the directory `/papers`, the agent records LCW information for the directory `/papers`, but no updates are made to $\mathcal{M}$.

$\mathcal{L}$ = {LCW(in.dir(f, /kr94)),
LCW(in.dir(f, /kr94) $\wedge$ size(f, l)),
LCW(in.dir(kr.tex, d)),
LCW(size(kr.tex, l)),
LCW(in.dir(kr.ps, d)),
LCW(size(kr.ps, l)),
LCW(in.dir(f, /papers)),
LCW(in.dir(f, /papers) $\wedge$ size(f, l))}

Moving the file `kr.ps` from the directory `/kr94` to the directory `/papers` results in both Domain Contraction to the directory `/kr94`, and Domain Growth to the directory `/papers`. $\mathcal{M}$ is updated as follows:

$\mathcal{M}$ = {in.dir(kr.tex, /papers),
size(kr.tex, 100),
in.dir(kr.ps, /kr94),
size(kr.ps, 300)}

There is no change to $\mathcal{L}$ due to Domain Contraction. However, Domain Growth could potentially result in statements being retracted from $\mathcal{L}$. This example illustrates the advantage of having the Domain Growth Rule retract the set of `MREL` sentences from $\mathcal{L}$, rather than naively retracting the set of `REL` sentences. There are three statements in `REL`:

$$\texttt{REL}(\texttt{in.dir}(\texttt{kr.ps}, \texttt{/papers})) = \{\textsc{lcw}(\texttt{in.dir}(f, \texttt{/papers})), \textsc{lcw}(\texttt{in.dir}(\texttt{kr.ps}, d)), \textsc{lcw}(\texttt{in.dir}(f, \texttt{/papers}) \wedge \texttt{size}(f, l))\}$$

However, the MREL of the update is empty, because we have LCW on the size of kr.ps:

$$\texttt{MREL}(\texttt{in.dir}(\texttt{kr.ps}, \texttt{/papers})) = \{\}$$

As a result, $\mathcal{L}$ remains unchanged after the mv command is executed. However, if we did not know the size of kr.ps when it was moved, we would have lost LCW on the size of the files in the directory /papers.

The last action in our example is compressing the file kr.ps. This action illustrates the advantage of retracting REL rather than PREL in the Information Loss Rule. After the file kr.ps is compressed, its size becomes unknown. Thus, $\mathcal{M}$ shrinks to:

$$\mathcal{M} = \{\texttt{in.dir}(\texttt{kr.tex}, \texttt{/papers}), \texttt{size}(\texttt{kr.tex}, 100), \texttt{in.dir}(\texttt{kr.ps}, \texttt{/kr94})\}$$

The set of PREL statements is:

$$\texttt{PREL}(\texttt{size}(\texttt{kr.ps}, l)) = \{\textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94}) \wedge \texttt{size}(f, l)), \textsc{lcw}(\texttt{in.dir}(f, \texttt{/papers}) \wedge \texttt{size}(f, l)), \textsc{lcw}(\texttt{size}(\texttt{kr.ps}, l))\}$$

In contrast, because we know that kr.ps is now in the directory /papers, the set of REL statements contains only the following:

$$\texttt{REL}(\texttt{size}(\texttt{kr.ps}, l)) = \{\textsc{lcw}(\texttt{in.dir}(f, \texttt{/papers}) \wedge \texttt{size}(f, l)), \textsc{lcw}(\texttt{size}(\texttt{kr.ps}, l))\}$$

Thus after the `compress` action is executed, we remove the `REL` statement from $\mathcal{L}$, obtaining:

$$\mathcal{L} = \{\textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94})), \textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94}) \wedge \texttt{size}(f, l)), \textsc{lcw}(\texttt{in.dir}(\texttt{kr.tex}, d)), \textsc{lcw}(\texttt{size}(\texttt{kr.tex}, l)), \textsc{lcw}(\texttt{in.dir}(\texttt{kr.ps}, d)), \textsc{lcw}(\texttt{in.dir}(f, \texttt{/papers}))\}$$

3.5.7 Computational Complexity of Updates

As stated earlier, our motivation for formulating conservative update rules has been to keep LCW update tractable. We make good on this promise below by considering the complexity of applying each update rule.

- **Information Gain:** The Information Gain Rule implies that no sentences have to be retracted from $\mathcal{L}$. LCW sentences may be added by the Information Gain Rule in time that is independent of the size of $\mathcal{L}$. The Counting Rule requires counting ground instances of a literal in $\mathcal{M}$, which requires time that is at most linear in the size of the database. For the most part, the Composition Rule is applied only in response to LCW queries; when applied proactively after action execution, it requires time that is linear in the number of atomic Δ updates, which correspond to the action's effects.

- **Information Loss:** First, the agent computes the set $\texttt{PREL}(\varphi)$, which takes time linear in the size of $\mathcal{L}$ in the worst case. Next, the agent computes $\texttt{REL}(\varphi)$ from $\texttt{PREL}(\varphi)$. Since this means determining whether $\mathcal{L} \wedge \mathcal{M} \vdash \neg(\Phi - \varphi)\theta$, the agent can incur an $O(|\mathcal{L}|^{c-1}|\mathcal{M}|^{c-1})$ cost for an LCW query (where c denotes the maximum number of conjuncts in a sentence in $\mathcal{L}$) for each element in $\texttt{PREL}$ (Theorem 2.9). Thus, to determine the worst case complexity of Information Loss, one must calculate the maximum length of the elements of $\mathcal{L}$. This is easy because there are only two ways that LCW sentences can be added to $\mathcal{L}$: via the Information Gain Rule or via proactive use of the Composition Rule. Since the first method only adds literals, $c = 1$. While the Composition Rule could (in theory) lead to an LCW sentence of arbitrary length, it is only used for forward chaining in a limited context (as explained in Section 3.5.2). As a result, it never adds sentences that are longer than a constant bound determined by the planning domain. For the UNIX and Internet domains, this constant is 3. In summary, $|\texttt{PREL}(\varphi)|$ is potentially linear in the size of $\mathcal{L}$, so computing $\texttt{REL}(\varphi)$ from $\texttt{PREL}(\varphi)$ could take $O(|\mathcal{L}|^{c}|\mathcal{M}|^{c-1})$. This cost dominates the time for the entire update.

- **Domain Growth:** The agent has to compute the set $\texttt{REL}(\varphi)$ which, as explained above, is polynomial in the size of $\mathcal{L}$ and $\mathcal{M}$. Computing $\texttt{MREL}(\varphi)$ from $\texttt{REL}(\varphi)$ is linear in the size of $\texttt{REL}$, but polynomial in the size of $\mathcal{L}$ and $\mathcal{M}$, since additional bounded-length LCW queries may be involved. The agent then removes each element of the set from $\mathcal{L}$, which takes time linear in the size of the set $\texttt{MREL}(\varphi)$. Thus the whole operation is $O(|\mathcal{L}|^{c}|\mathcal{M}|)^{c-1})$.

- **Domain Contraction:** $\mathcal{L}$ remains unchanged in this case.

We summarize the preceding discussion with the following theorem.

Theorem 3.18 (Tractability of Updates) *Let $\mathcal{M}$ be a set of ground literals, and let $\mathcal{L}$ be a set of positive conjunctive* LCW *sentences such that no member of $\mathcal{L}$ has more than c conjuncts. Updating $\mathcal{L}'$ in response to an atomic change requires time that is at most $O(|\mathcal{L}|^c|\mathcal{M}|)^{c-1})$.*

The use of standard indexing techniques (*e.g.*, hashing on the predicates in φ) renders the effective polynomial coefficient lower than the conservative bound we present.

3.5.8 Optimality of Atomic Update Policies

Since sentences in $\mathcal{L}$ are restricted to positive conjunctions, and since our update rules are conservative, the update process is incomplete. Nevertheless, it is easy to see that our algorithm is better than the trivial update algorithm ($\mathcal{L}' \leftarrow \{\}$). In our softbot's domain, for example, the Information Gain and Counting Rules enable us to derive LCW from a wide range of "sensory" actions, including `pwd, wc, grep, ls, finger`, and many more. Furthermore, our update rules retain LCW in many cases. For example, changes to the state of one "locale" (such as a directory, a database, an archive, etc.) do not impact LCW on other locales. This feature of our update calculus applies to physical locales as well.

Below, we make a much stronger claim, that the sets of sentences retracted by theorems 3.15 through 3.17 are, in fact, minimal. Every sentence retracted is invalid and *must* be removed from $\mathcal{L}$ to maintain soundness. This statement is trivially true for Domain Contraction where no sentences are retracted. Clearly, we cannot do better than that. The following theorem asserts that each LCW sentence retracted due to Information Loss is, in fact, invalid.

Theorem 3.19 (Minimal Information Loss) *Let $\mathcal{M}$, $\mathcal{L}$ be a conservative representation, and let φ be a positive literal. Let A denote an atomic change of the form*

$\Delta(\varphi, \mathtt{T} \rightarrow \mathtt{U})$ *or of the form* $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{U})$. *If* $\Phi \in \mathtt{REL}(\varphi)$ *then* LCW*(*Φ*) does not hold after A has occurred.*

The corresponding result holds for Domain Growth.

Theorem 3.20 (Minimal Domain Growth) *Let* $\mathcal{M}$*,* $\mathcal{L}$ *be a conservative representation, and let* φ *be a positive literal. Let A denote an atomic change of the form* $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{T})$. *If* $\Phi \in \mathtt{MREL}(\varphi)$ *then* LCW*(*Φ*) does not hold after A has occurred.*

Are the update rules for Information Loss and Domain Growth the best possible? At first blush, the answer to this question would seem to be yes, since the rules are sound and they retract the minimal set of sentences from $\mathcal{L}$. So what more could we want? However, this observation overlooks the key fact that inference in our framework is lazy, so that when the sentence φ is retracted we effectively also retract $\varphi\sigma$ for any variable substitution σ. Above, we claimed that the sentence φ really ought to be retracted, but we didn't claim that the sentence $\varphi\sigma$ (which is weaker!) is invalid. In fact, there are cases where such sentences are valid. For example, consider the case where we have LCW on the size of all the files in the directory `/bin`, but then the file `a.out` in that directory is compressed. Our update rule for Information Loss would retract the LCW statement, when, in fact, a weaker statement that we know the size of all the files in `/bin` — except `a.out` — is true. Since the sentences in $\mathcal{L}$ are conjunctions of positive literals, we have no way of expressing the above statement in a single LCW formula; thus, the original formula would have to be replaced with many new formulas.

3.5.9 Optimal Order of Atomic Updates

So far our discussion has been restricted to atomic updates, but many updates consist of a set of such atoms. While the order in which these atomic components are handled does not affect the eventual contents of $\mathcal{M}$, this is not true for $\mathcal{L}$. Of course the $\mathcal{M}$, $\mathcal{L}$

pair will be a conservative representation of $\mathcal{S}$ regardless of the order chosen, but some orderings will discard more sentences from $\mathcal{L}$ than others. For example, consider the following imaginary UNIX command `gen-file` d which creates a new file in directory d and gives it zero size. Say the effects of executing `gen-file /tex` are as follows:

$\Delta(\texttt{in.dir(G003, /tex)}, \texttt{F} \rightarrow \texttt{T})$
$\Delta(\texttt{size(G003, } n\texttt{)}, \texttt{U} \rightarrow \texttt{T} \vee \texttt{F})$

Now, suppose that before executing `gen-file /tex` the agent knew the of all files in `/tex`.

$$\text{LCW}(\texttt{in.dir(}f\texttt{, /tex)} \wedge \texttt{size}(f, c))$$

If the atomic updates are processed in the order given, then the Domain Growth Rule will eliminate this LCW sentence from $\mathcal{L}$, but if the updates are processed in the reverse order then that retraction is unnecessary.

To obtain an optimal order, an agent must be sure that as many sentences are added to $\mathcal{L}$ as possible and that as few are removed as possible. We believe the following order suffices:

1. Process Domain Contraction updates.

2. Process Information Gain updates and apply the Counting Rule.

3. Process Domain Growth updates.

4. Process Information Loss updates.

The insight behind this order is as follows. The only types of updates that remove items from $\mathcal{L}$ are Domain Growth and Information Loss, which remove the sets $\texttt{MREL}(\varphi)$ and $\texttt{REL}(\varphi)$ respectively. More information present in $\mathcal{M}$, $\mathcal{L}$ means that we'll have more LCW information and be able to prove more sentences are false. This in

turn means that the REL and MREL sets will be as small as possible. So Information Gain and Domain Contraction updates should be processed first.[18] It can be useful to process Domain Contraction Updates before Information Gain because this improves the chance that the proactive application of the Composition Rule (Section 3.5.2) will result in new LCW sentences.

3.6 Summary

We introduced SADL, a language for representing sensing actions and information goals.

1. Since knowledge goals are temporal, SADL supports the temporal annotation **initially**.

2. In *knowledge-free Markov domains*, such as UNIX, knowledge preconditions for actions are inappropriate, but subgoaling to obtain knowledge is often necessary; SADL handles this paradox by eliminating knowledge preconditions from actions, but using secondary preconditions to clearly indicate when subgoaling to acquire knowledge could be useful.

We described the connections between SADL and LCW, by

1. Explaining how SADL supports LCW goals, including goals tagged with **initially**.

2. Explaining how SADL effects can result in updates to LCW, and providing sound, tractable procedures for performing these updates to LCW.

In the next chapter, we discuss how the actions presented in this chapter are combined to form plans, and we introduce the structures used by the PUCCINI planning algorithm.

---

[18] The only problem with this argument would arise if a Domain Growth or Information Loss update removed an LCW sentence that had been added by Information Gain, but this is impossible because we assume that every set of updates corresponding to a single action or event is mutually consistent.

Chapter 4

PLANS AND PLANNING PROBLEMS

4.1 Partially specified plans

We have described plans as totally-ordered sequences of ground actions, but when building plans incrementally, it's useful to have a less rigid specification of a plan. We describe plans instead as partially-ordered sequences of partially-instantiated action schemas.

Planning can be regarded as a search, through a space of partially-specified plans, for a complete plan that achieves the goal. A PUCCINI plan consists of a tuple $\langle \mathcal{A}, \mathcal{O}, \mathcal{B}, \mathcal{C}, \mathcal{E} \rangle$, where $\mathcal{A}$ is of a set of *steps* (partially instantiated action schemas), $\mathcal{O}$ is set of ordering constraints, representing a partial *temporal ordering relation* over $\mathcal{A}$, $\mathcal{B}$ is a set of variable binding constraints, representing a a partial equivalence relation over the variables and constants in the plan, $\mathcal{C}$ is a set of *causal links* [84] and $\mathcal{E}$ is a set of step execution labels, representing a function mapping each step to its execution status.

4.1.1 Steps and Effects

Steps are structures representing partially specified actions. Semantically, they are sets of actions, containing all ground actions consistent with the action schema and the binding constraints in $\mathcal{B}$, which will be narrowed down to singletons prior to execution. Thus we can talk about executing steps or executing actions interchangeably. Since steps are sets of actions, we will represent them using uppercase A. We define a step formally as follows.

Definition 4.1 *A step A is a tuple $\langle \pi, \varepsilon, \theta, \mathsf{Proc} \rangle$, where π_A and ε_A (preconditions and effects of A) are lifted versions of the action preconditions and effects π_a and ε_a introduced in Section 3.3. θ_A is the set of equality and inequality constraints on free variables in π_A, and Proc_A is a procedure for executing A and returning the sensed information to the planner.*

Recall that conditional effects are defined in Section 3.4.1 as sentences of the form **when** $\gamma_P^{\mathtt{T}}(a)$ **cause**(P, $\mathtt{T}$). These are represented internally as planner structures.

Definition 4.2 *An effect ε is a triple $\langle \gamma_P, P, \theta \rangle$, where γ_{P_ε} is the (lifted) effect precondition of ε, P_ε is the outcome of ε, and θ_ε the set of equality and inequality constraints on free variables in γ_{P_ε}.*

4.1.2 Ordering constraints

$\mathcal{O}$ consists of pairwise ordering constraints, using two ordering relations: The intransitive *successor* relation ";" and the transitive closure of ";", "$\prec$". $A_a ; A_b$ means that in any total ordering $\mathcal{A}_t$ of $\mathcal{A}$, if $\mathcal{A}_t[i] = A_a$ then $\mathcal{A}_t[i+1] = A_b$. "$\prec$" is the customary ordering relation used by partial-order planners. $A_a \prec A_b$ means that in any total ordering of steps $\mathcal{A}_t$, if $\mathcal{A}_t[i] = a$ and $\mathcal{A}_t[j] = b$, then $i < j$. The ordering of steps corresponds to the temporal order in which they are executed. The reason for the relation ";" is that steps are executed during planning, which represents a stronger commitment to order than "$\prec$" alone can express. That is, any executed step is necessarily before all unexecuted steps, including steps that haven't been added to the plan yet. We can ensure that no actions will be ordered before the executed prefix $A_1 \ldots A_e$ by imposing the constraints $A_0 ; A_1 ; \ldots ; A_e$. We say that $\mathcal{O}$ is consistent if there is some permutation of $\mathcal{A}$ that doesn't violate any of the ordering constraints imposed by $\mathcal{O}$.

4.1.3 Binding constraints

$\mathcal{B}$ consists pairwise binding constraints using two binding relations: The codesignation relation $\approx$, and the non-codesignation relation $\not\approx$; by definition, if a and b are distinct constants, $a \not\approx b$. $\mathcal{B}$ is consistent if there is some assignment of constants to variables that obeys all of these constraints.

4.1.4 Causal links

A causal link is a tuple $\langle A_p, e, q, A_c, t_1, t_2 \rangle$, consisting of a record of the planner's decision to support precondition q of step A_c with an effect e of step A_p, and a *protection interval* $[t_1, t_2]$ over which condition q must be preserved. Typically, $[t_1, t_2]$ is just $[A_p, A_c]$. A_c is known as the *consumer* of q and A_p is known as the *producer* of e. We will use the notation $t_1 \xrightarrow{e,q} t_2$ to represent links; it will always be obvious who the producer and consumer are, since these are the actions with effect e and precondition q, respectively.

We use *causal* link as a generic term, even though links are not only used to record causal support; they are also used to record observational support, LCW support, and even assumptions. We also use the terms *observational link*, LCW *link*, $\forall$ *link*, etc.

Once the causal link is added, the planner is committed to ensure that condition q remains true over the protection interval. A step A_t that could possibly be executed during this interval and might make q false is said to *threaten* the link, and the planner must make take evasive action — for example, by ensuring that A_t is executed after A_c. Techniques for handling threats are discussed in Section 5.3

4.1.5 Step execution

In the simplest case, $\mathcal{E}$ maps each step in the plan to `executed` or `unexecuted`, though other values, such as `executing` can be used to give a finer level of information of the status of steps. Knowing that a step is `executing` would be useful when

executing multiple steps simultaneously, for example to handle resource conflicts [43]. By definition, $\mathcal{E}(A_0)$ is always **executed**, and when $\mathcal{E}(A_\infty) =$ **executed**, planning is complete.

4.1.6 Plan refinement and repair

Planning proceeds by making incremental additions to previous plans, until a complete plan is generated. The normal planning process only adds structures to a plan, in a process known as *refinement*, thus reducing the set of complete plans consistent with the partial plan:

Definition 4.3 *A plan* $R = \langle \mathcal{A}_R, \mathcal{O}_R, \mathcal{B}_R, \mathcal{C}_R, \mathcal{E}_R \rangle$*, is a refinement of a plan* $P = \langle \mathcal{A}_P, \mathcal{O}_P, \mathcal{B}_P, \mathcal{C}_P, \mathcal{E}_P \rangle$ *if and only if* $\mathcal{A}_P \subseteq \mathcal{A}_R$*,* $\mathcal{O}_P \subseteq \mathcal{O}_R$*,* $\mathcal{B}_P \subseteq \mathcal{B}_R$*,* $\mathcal{C}_P \subseteq \mathcal{C}_R$ *and* $\mathcal{E}_P \subseteq \mathcal{E}_R$*.*

When all we are concerned with is plan generation, refinement is sufficient to explore all plans that could achieve the goal. However, when these plans are being executed in the course of planning and both the state of the world and the agent's knowledge are changing, the idea of *plan repair* becomes useful. In the course of plan repair, structures are removed from the plan, and no structures are added:

Definition 4.4 *A plan* $R = \langle \mathcal{A}_R, \mathcal{O}_R, \mathcal{B}_R, \mathcal{C}_R, \mathcal{E}_R \rangle$*, is a repair of a plan* $P = \langle \mathcal{A}_P, \mathcal{O}_P, \mathcal{B}_P, \mathcal{C}_P, \mathcal{E}_P \rangle$ *if and only if* $\mathcal{A}_R \subseteq \mathcal{A}_P$*,* $\mathcal{O}_R \subseteq \mathcal{O}_P$*,* $\mathcal{B}_R \subseteq \mathcal{B}_P$*,* $\mathcal{C}_R \subseteq \mathcal{C}_P$ *and* $\mathcal{E}_R \subseteq \mathcal{E}_P$*.*

4.2 Interleaving Planning and Execution

In the classical planning framework, plans are constructed completely by the planner before they are executed, and there is no feedback between the planner and the environment. When planning with incomplete information, the planner must obviously take into account the result of sensing actions. This can be done within the classical

planning framework by adding branches to the plan to decide what course of action to take based on the result of the sensors, and producing complete sub-plans for all possible sensor values.

While this *contingency planning* approach is formally clean, it has a number of drawbacks. Because the plans produced can be exponential in the size of the problem, and the planning problem itself already requires searching an exponentially large space of plans, contingency planners have a double exponential time complexity. Furthermore, since there is no bound on the number of values that can be sensed by an action with universally quantified observational effects, contingent plans can have an infinite branching factor. Also, as we've discussed, actions like finding a file or a web page can involve an unbounded amount of search, so there's no way, in the absence of using loops, to decide when to stop planning.

A more practical approach is to *interleave planning with execution* (IPE), thus finding information needed to determine which branch to follow (before generating all options), and to determine when planning should stop. We can view the plans produced by interleaving planning with execution in terms of contingency plans. Intuitively, such a plan follows one branch of the corresponding contingent plan, by deferring the planning for contingencies until the information needed to make the correct choice is known.

4.3 The Planning Problem

The traditional definition of a planning problem is a tuple consisting of a set of allowable actions, a complete description of the initial state, and a goal. Given that our agents have incomplete information about the world, we replace the complete description of the initial state with an incomplete theory of the initial state. The agent cannot consult the actual state of the world directly, but it does interact with the world, and thus can acquire information about the true state indirectly, through

sensing. Thus, the world state acts as a kind of oracle, which the agent can query in a restricted way.

Recall that, in Chapter 2, we described the agent's knowledge as part of the state, using the predicate K. While that makes the formalism simpler, when we actually build the agent, we need to separate the agent's knowledge from the state itself, since its knowledge is all that can be represented explicitly (in our case, using the $\mathcal{L}$ and $\mathcal{M}$ databases). The actual initial state of the world (s_i), not including the agent's knowledge, is just the situation s_0 but without the predicate K. The agent's incomplete theory of the initial state, which we will denote $\mathcal{I}$, consists of those facts that can be inferred from s but not from s_i.

Definition 4.5 *A* planning problem *is a tuple* $\langle \mathcal{D}, s_i, \mathcal{I}, \Gamma \rangle$, *where* $\mathcal{D}$ *is a theory of action,* s_i *is the initial state of the world (which is possibly unknown to the agent),* $\mathcal{I}$ *is an incomplete theory of the world, which is consistent with* s_i *and* Γ *is a goal given to the planner to solve.*

Our definition of a solution to a planning problem is similar to the classical definition. Note, however, that the agent need not know prior to executing the plan that the plan is indeed a solution. The agent must know *after* execution that it has succeeded, but that follows from the definition of the SADL goal annotations. We separate the state of the world, s_i from the agent's knowledge in $\mathcal{I}$ to make it clear how the two inputs are used, but for simplicity, we will still refer to their combination, s_0.

Definition 4.6 *A sequence of actions,* $\{a\}_1^n$, *is a* solution *to a planning problem* $\langle \mathcal{D}, s_i, \mathcal{I}, \Gamma \rangle$ *iff*

1. $\{a\}_1^n$ *consists entirely of actions instantiated from* $\mathcal{D}$

2. $\{a\}_1^n$ *is executable from* s_0 *(see Section 3.4.3).*

3. and $DO(\{a\}_1^n, s_0) \models \Gamma$.

The use of the actual situation s_i is key, since what counts is that the goal is satisfied when the plan is executed from s_i, not that the agent can prove it will succeed before it begins execution. Finding an action sequence that is guaranteed *a priori* to work in all situations s consistent with $\mathcal{I}$ is in general impossible. However, since the agent can consult s_i as an oracle in the course of planning, it can produce a plan that does in fact work for the particular situation s_i.

Definitions such as 4.6 are usually applied to planners that produce a single, correct plan before executing it. But using IPE, the planner might blunder along, executing the prefixes of many "plans" before finding the correct one. The above definition applies to IPE, but we must be more specific about what $\{a\}_1^n$ refers to. Since the planner may execute the prefix of more than one plan, only considering the action sequence of the last plan visited is too narrow. This is a particular problem for **initially** and **hands-off** goals, since the entire history of action execution is relevant. Thus, $\{a\}_1^n$ refers to the entire sequence of actions generated by the planner, regardless of what "plans" these actions correspond to.

4.4 Summary

In this chapter, we presented the structures used to represent plans and planning problems based on the action language presented in the previous chapter. In the next chapter, we discuss the PUCCINI planning algorithm, which constructs plans using these structures.

Chapter 5

PUCCINI ALGORITHM

In this chapter, we describe the PUCCINI algorithm. PUCCINI [1] is based on our XII planner [35], which in turn is based on UCPOP [72], a classical planner that supports a subset of ADL [71]. PUCCINI extends UCPOP by handling incomplete information and LCW, interleaving planning with execution, and handling all of the SADL extensions to ADL.

PUCCINI is given a planning problem as input, and executes a sequence of actions, returning success if that sequence consists of a solution to the planning problem. We first provide a high-level overview of the PUCCINI algorithm. We then discuss how PUCCINI achieves preconditions of actions, including a powerful mechanism known as *assumption*. We then discuss how PUCCINI prevents subgoal clobbering and finally how it interleaves planning with execution. PUCCINI is sound, but incomplete. In the next chapter, we prove soundness and discuss the incompleteness of PUCCINI.

5.1 Planning overview

The input to PUCCINI is a planning problem (see Chapter 4), $\langle \mathcal{D}, s_0, \mathcal{I}, \Gamma \rangle$. This problem is encoded as a "dummy plan," P_0, which the PUCCINI algorithm refines into a complete solution by incremental additions to the plan in the form of steps, causal links, binding constraints, etc. This plan consists of two "dummy" steps: the *initial step*, A_0, whose effects represent $\mathcal{I}$, and the *goal step*, A_∞, whose precondition is Γ. Using this representation simplifies the algorithm, since the same code used to

[1] PUCCINI stands for Planning with Universal quantification, Conditional effects, Causal links, and INcomplete Information

reason about preconditions and effects of actions can be used to reason about goals and initial conditions. The effects of A_0 are not stored explicitly in the step structure, as the size of $\mathcal{I}$ can be quite large. Rather, the planner consults the databases $\mathcal{M}$ and $\mathcal{L}$ discussed in Chapter 2 whenever effects of A_0 are requested. Furthermore, since the agent's knowledge of the initial state changes over time, $\mathcal{I}$ is not static. The agent's knowledge about the current situation also changes over time, but this additional information is reflected in the effect structures of the plan itself. The only time the agent's knowledge of the current state needs to be consulted explicitly is after backtracking over execution (see Section 5.4.2 for details).

Literals in $\mathcal{I}$ have the standard truth values T, F and U, as well as a fourth truth value, I, meaning inaccessible. A literal labeled with I has been changed prior to being sensed, so it is impossible to find out what truth value it originally had. There is also LCW knowledge associated with $\mathcal{I}$, but the only LCW knowledge the agent keeps track of is *current* LCW knowledge about what held in the initial state, which is weaker (and more useful) than knowledge about LCW statements that were valid in s_0. That is, the agent can record LCW(**initially**(φ)), but not **initially**(LCW(φ)). **initially**(LCW(φ)) means the agent had LCW knowledge over φ in the initial state. Since knowledge about the initial state never goes away (Lemma 6.3), **initially**(LCW(φ)) $\Rightarrow$ LCW(**initially**(φ)).

The initial call to PUCCINI (Figure 5.1) is PUCCINI(P_0, $\langle \Gamma, A_\infty \rangle$, $\mathcal{D}$, s_0), where $\langle \Gamma, A_\infty \rangle$ is the initial contents of the *goal agenda*, $\mathcal{G}$, which contains pairs $\langle g, S \rangle$, consisting of a goal condition g and a step S for which g is a precondition.

In subsequent recursive calls: PUCCINI(P, $\mathcal{G}$, $\mathcal{D}$, s), PUCCINI refines plan P, adding to and removing from $\mathcal{G}$ as appropriate, and changing the situation s by executing actions. As conditions are satisfied, they are removed from $\mathcal{G}$, and when actions are added to the plan, preconditions of those actions are added to $\mathcal{G}$. PUCCINI terminates with success if P is complete and consistent, $\mathcal{G}$ is empty, and all steps in $\mathcal{A}$ are labeled as executed in $\mathcal{E}$. PUCCINI returns failure (and backtracks) if constraints in $\mathcal{O}$ or $\mathcal{B}$

become inconsistent, a required condition is revealed to be false after executing an action, or a condition on $\mathcal{G}$ cannot be achieved.

5.1.1 Searching through Plan-Space

Generative planning can be viewed as a process of gradually refining a partially-specified plan, until it is transformed into a complete plan that solves the goal.

We refer to the aspects of the plan that need to be changed before the plan can be complete as *flaws*, and the changes made as *fixes*. In traditional least-commitment planners, the flaws to consider are open goals (goals on the goal agenda $\mathcal{G}$), and threats to causal links. In order to interleave planning and execution in this framework, we treat an unexecuted action as another type of flaw, whose fix is to execute it (Section 5.4). Treating execution in this manner has some profound impacts on the search algorithm, as we discuss in depth in Section 5.4. Once no more flaws exist, the plan is complete (and fully executed), and the goal is achieved.

PUCCINI($\langle\mathcal{A},\mathcal{O},\mathcal{B},\mathcal{C},\mathcal{E}\rangle$, $\mathcal{G}$, $\mathcal{D}$, s)

1. If $\mathcal{G}=\emptyset \wedge \mathcal{E}=\emptyset \wedge \forall\ell\in\mathcal{C}\ \ell$ not threatened, return success.
2. Pick one of
   (a) HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$, $\mathcal{D}$)
   (b) s = HandleExecution($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{E}$, s)
3. HandleThreats($\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$)
4. PUCCINI($\langle\mathcal{A},\mathcal{O},\mathcal{B},\mathcal{C},\mathcal{E}\rangle$, $\mathcal{G}$, $\mathcal{D}$, s)

Figure 5.1: PUCCINI Algorithm

In general, there will be more than one fix for a given flaw, and for completeness, all fixes must be considered. Considering all fixes requires search, but we can view

the algorithm in terms of nondeterministic choice by assuming that for each flaw, PUCCINI miraculously chooses the correct fix. Using nondeterminism simplifies the algorithm description, and allows us to separate the details of the planner from the details of the search strategy. However, it conceals the phenomenon of backtracking over execution (Section 5.4.2) because a nondeterministic planner will never make choices requiring it to backtrack.

In the rest of this chapter, we elaborate on some novel elements of the PUCCINI search space. Sections 5.2, 5.3 and 5.4 discuss how PUCCINI handles the flaws of open goals, threats and unexecuted steps, respectively.

5.2 Handling Open Goals

A goal is said to be "open" if it is still on the goal agenda. An open goal in PUCCINI can be any arbitrary goal expression in the SADL language, including disjunction, conjunction, and nested quantification. To cope with this complexity, PUCCINI breaks the goals apart into simpler goals using a divide-and-conquer strategy.

5.2.1 Canonical Form

The key to this divide-and conquer strategy is a preprocessing step that converts all preconditions and effects into a canonical form. With the exception of LCW conditions (see Section 5.2.6), all preconditions and postconditions are converted into a uniform representation in which the atomic elements are free of conjunction and disjunction, but may involve universal quantification. Each atomic element is a literal in the SADL language consisting of an annotation such as **initially** or **cause**, a truth value of `T`, `F`, `U` or a variable, and a *condition*, consisting of a predicate with its arguments, any of which may be a universally quantified variable. We call these atomic elements *alits* for annotated literals. Matching is done between alits using simple unification (see Section 5.2.4).

Existentials are replaced with Skolem functions. The scope of universal quantifiers becomes global and the scope of negation is restricted to individual literals. For conditional effects, the scope of the precondition is also restricted to individual literals. The effect of this conversion is that all expressions consist of alits, possibly with preconditions, connected by $\wedge$ and $\vee$. Since disjunction is not allowed in effects, all effects, which become conjunctions of alits, are conceptually equivalent to STRIPS add and delete lists. The conjunctions and disjunctions in preconditions are handled within the planning algorithm, where conjuncts are added as separate elements to the goal agenda $\mathcal{G}$, and a disjunction is treated as a nondeterministic choice (Lines 3 and 4 of Reduce, Figure 5.3) . The result is that atomic goals on $\mathcal{G}$ are alits, and these are satisfied by alits on the add/delete list.

The canonical form for universally quantified goals is a little more involved. SADL places no restrictions on the form of $\forall$ goals, and does not require an explicit universe of discourse to be specified. However, being able to obtain LCW on a conjunctive universe provides a powerful way of solving $\forall$ goals. In order to exploit this, PUCCINI converts $\forall$ goals, when possible, into a form $\forall \vec{x}\ (P(\vec{x}) \Rightarrow Q(\vec{x}))$, where the $P(\vec{x})$ is referred to as the *context* of the goal. As we will see, the context can be used in other interesting ways as well. In order to compute the context, PUCCINI converts the goal into DNF, and moves all the single negated disjuncts over to the left-hand side of the implication symbol.

5.2.2 The Universal Base

Like other planners, PUCCINI makes use of the *universal base* [86] Υ, a universally ground conjunctive formula in which universally quantified variables have been replaced by each possible constant in the universe of discourse. Obviously, this depends on the universe of discourse being finite and known to the agent, two assumptions that most planners depend on. As we discuss in Section 5.2.5, we don't assume that either condition holds.

HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$, $\mathcal{D}$)

if $\mathcal{G} \neq \emptyset$ then pop $\langle$g, $S_c\rangle$ from from $\mathcal{G}$ and select case:

1. If g = Context $\Rightarrow$ $cond$, and Context$\models$ $cond$ then g is trivially satisfied.
2. If g is a **hands-off** goal, then call AddLink(A_0, g, nil, S_c, $\mathcal{O}$, $\mathcal{B}$)
3. Else nondeterministically choose
   (a) Reduce(g, $\mathcal{G}$)
   (b) Instantiate a new action A_{new} from $\mathcal{D}$, such that MGU(e, g) $\neq\perp$, and add it to $\mathcal{A}$. Call Addlink(A_{new}, g, e, S_c, $\mathcal{O}$, $\mathcal{B}$). Add preconditions of A_{new} to $\mathcal{G}$.
   (c) Choose an existing action A_{old} from $\mathcal{A}$, such that MGU(e, g) $\neq\perp$, and Call Addlink(S_p, g, e, A_{old}, $\mathcal{O}$, $\mathcal{B}$).

Figure 5.2: Procedure HandleGoal

The universal base, Υ, is a function from goals, possibly including quantifiers, to quantifier-free goal expressions. The idea is quite simple. For example, given the goal of putting all blocks on a table, if there are three blocks in the universe, named A, B and C, then the universal base would consist of the goal of putting A on the table, putting B on the table and putting C on the table. Formally speaking, the universal base replaces universally quantified variables with their extensions and replaces existentially quantified variables (which are represented as Skolem functions) with Skolem constants. We define the universal base as follows:

$$\begin{aligned}
\Upsilon(\forall x.f(x) \Rightarrow g(x)) &= g(x_1) \wedge \ldots \wedge g(x_n), \text{for each } x_i \text{ such that} f(x). \\
\Upsilon(\exists x.g(x)) &= g(x'), \text{where } x' \text{ is a Skolem constant} \\
\Upsilon(R(x_1 \ldots x_n)) &= R(x_1 \ldots x_n) \\
\Upsilon(\bigwedge_{i=1}^{n} \varphi_i) &= \Upsilon(\varphi_1) \wedge \ldots \wedge \Upsilon(\varphi_) \\
\Upsilon(\bigvee_{i=1}^{n} \varphi_i) &= \Upsilon(\varphi_1) \vee \ldots \vee \Upsilon(\varphi_)
\end{aligned} \tag{5.1}$$

Reduce(g, S_c)

1. If g is an LCW goal, then nonderministically choose one of the following
   (a) Choose Φ and Ψ, such that g $= \text{LCW}(\Phi \wedge \Psi)$. Add $\langle \text{LCW}(\Phi), S_c \rangle$ to $\mathcal{G}$ and add $\langle \forall \vec{x}.\Phi \Rightarrow \text{LCW}(\Psi)), S_c \rangle$ (where $\vec{x}$ is the set of free variables appearing in Φ) to $\mathcal{G}$, tagged to wait on the result of $\text{LCW}(\Phi)$.
   (b) Choose Φ and Ψ, such that g $= \text{LCW}(\Phi \wedge \Psi)$. Add $\langle \text{LCW}(\Phi), S_c \rangle$ and $\langle \text{LCW}(\Psi), S_c \rangle$ to $\mathcal{G}$.
2. If g is of the form $\forall \vec{x}.\Phi \Rightarrow \Psi$ then nonderministically choose one of the following
   (a) If $\text{LCW}(\Phi)$ is true, replace g with the universal base Υ(g). Otherwise, add $\langle \text{LCW}(\Phi), S_c \rangle$ to $\mathcal{G}$, and add $\langle \forall \vec{x}.\Phi \Rightarrow \Psi, S_c \rangle$ to $\mathcal{G}$, tagged to wait on the result of $\text{LCW}(\Phi)$.
   (b) Partition: Nondeterministically choose some expression ϖ.
   Add $\langle \varpi \Rightarrow g, S_c \rangle$ to $\mathcal{G}$.
   Add $\langle \neg\varpi \Rightarrow g, S_c \rangle$ to $\mathcal{G}$.
3. If g $= g_1 \wedge g_2 \wedge \ldots \wedge g_n$: Add all $g_1, g_2, \ldots, g_n$ to $\mathcal{G}$
4. g $= g_1 \vee g_2 \vee \ldots \vee g_n$: Nondeterministically add one of $g_1, g_2, \ldots, g_n$ to $\mathcal{G}$

Figure 5.3: Procedure Reduce

5.2.3 Atomic Goals

An atomic goal is a goal that is free of conjunction and disjunction. It may, however, involve universal quantification. We explain how universally quantified goals are handled in Section 5.2.5.

Atomic goals can be satisfied by one of two means: adding a new action or committing to use the effect of an action already in the plan to support the goal (Lines 3(b,c) of HandleGoal, Figure 5.2). In the latter case, if the action supporting the goal is A_0, then the precondition is already satisfied, and we are simply adding constraints to ensure that it remains satisfied (see Figure 5.4). In PUCCINI, the contents of the agent's knowledge base, and hence the effects of A_0, can change during planning, so the search space can change dynamically, as we discuss in Section 5.4.2. If the goal is annotated with **initially**, then the protection interval of the link extends from A_0 to the producer S_p. Otherwise, the protection interval extends from producer S_p to consumer S_c. For example, if the goal is to know what file was initially named `core`, the name of the file must be left unchanged until it is sensed by S_p, but may be changed afterward. On the other hand, if the goal is to know what file is ultimately named `core`, then changing it before S_p is fine, but changing it afterward would invalidate the answer given by S_p.

It is easy to tell whether an effect satisfies an atomic goal. Since both effects and atomic goals are alits, determining whether an effect satisfies a goal reduces to the problem of matching two alits.

5.2.4 Matching

Matching between atomic preconditions and postconditions is fairly simple. The precondition of a conditional effect is not considered when matching that effect to a goal, so we need only match between alits, which consist of annotation, literal and truth value. We introduce the function MGU(e, p) (Figure 5.5), which returns the

Addlink(S_p, goal, eff, S_c, $\mathcal{O}$, $\mathcal{B}$)

(goal = Context $\Rightarrow g$; eff = **when**(p) e)

- If g is an **initially** goal, add $A_0 \stackrel{e,goal}{\rightarrow} S_p$ to $\mathcal{C}$. Otherwise, add $S_p \stackrel{e,goal}{\rightarrow} S_c$ to $\mathcal{C}$.
- Unless g is unannotated and e is an **observe** effect
  1. Add $S_p \prec S_c$ to $\mathcal{O}$
  2. If e is supported by an assumption link, whose producer is A_a, add $A_a \prec S_c$ to $\mathcal{O}$
- Add MGU(e, g) to $\mathcal{B}$.
- Add $\langle$Context $\Rightarrow p$, Sprod$\rangle$ to $\mathcal{G}$

Figure 5.4: Procedure Addlink

MGU(e, g)

1. If there is a consistent mapping θ of terms in e to terms in g, such that $e\theta$ and $g\theta$ are identical, then return that assigment.
2. If $e = (A_1(P(\vec{x}), v_1))$ and $g = A_2(Q(\vec{y}), v_2))$, and either
   (a) $A2 =$ **satisfy** OR
   (b) $A1 =$ **observe**

   then return MGU$(P(\vec{x}, v_1), (Q(\vec{y}, v_2)))$
3. If $e = \text{LCW}(\psi_1 \wedge \ldots \wedge \psi_n))$, and $g = \text{LCW}(\varphi_1 \wedge \ldots \wedge \varphi_n)$ and there is some permutation $j_1 \ldots j_n$ such that MGU$((\psi_{j_1}, \ldots \psi_{jn}), (\varphi_i, \ldots, \varphi_n))$, return that assignment
4. Else, return $\perp$

Figure 5.5: Most General Unifier

most general unifier, of e and p, meaning the minimal variable bindings to ensure that e and p will unify. If e and p can't unify, MGU(e, p) returns $\perp$.

In the course of unifying two variables, v_e from an effect, with v_g from a goal, we must consider the case in which one or both of them are universally quantified. Since a universally quantified goal can never be satisfied by a non-universally quantified effect, there are only two cases we need to consider:

- When both variables are universally quantified, we match them using MGU, but any codesignation constraints imposed by MGU between v_g and other variables in p are adopted as preconditions of e. Constraints are not posted to $\mathcal{B}$, which only records existential constraints

- When only v_e is universally quantified, the match succeeds without imposing any constraints on v_e. Constraints involving v_g are existentialized and added to $\mathcal{B}$.

5.2.5 *Universally Quantified Goals*

The effects of many UNIX commands are not easily expressed without $\forall$. The ubiquitous UNIX wildcard `*` can make almost any command universally quantified, and many other commands, such as `ls`, `ps`, `lpq`, and `rm` have $\forall$ effects.

However, the approach used by planners like UCPOP [72] and PRODIGY [62] to solve $\forall$ goals depends on the CWA, which presents a challenge for PUCCINI. Traditionally, planners satisfy $\forall$ goals by subgoaling to achieve the universal base (Equation 5.1), but this strategy relies on the closed world assumption. Only by assuming that all members of the universe of discourse are known (*i.e.*, represented in the agent's knowledge base) can one be confident that the universal base is equivalent to the $\forall$ goal. Since the presence of incomplete information invalidates the closed world assumption, the PUCCINI planner uses three new mechanisms for satisfying $\forall$ goals:

1. Sometimes it is possible to directly support a $\forall$ goal with a $\forall$ effect, without expanding the universal base. For example, given the goal of having all files in a directory group readable, PUCCINI can simply execute `chmod g+r *`; it doesn't need to know which files (if any) are in the directory. Because unification between $\forall$ variables is handled during matching, as we discussed in Section 5.2.4, supporting a $\forall$ goal with a $\forall$ effect is just a special case of supporting an atomic goal with an effect from a new or existing action. (Note that since universally quantified effects cannot appear in the agent's knowledge base, supporting a $\forall$ goal directly with effects of A_0 is not an option.) The only difference in the case of $\forall$ goals is that matching is between universally quantified expressions, and the causal link is labeled with a universally quantified expression. If the effect is conditional, and the precondition involves a $\forall$ variable, then a universally quantified precondition is added to $\mathcal{G}$.[2] However, if the context of the goal (introduced in Section 5.2.1) entails the precondition of the effect (or part of it) then that part of the precondition will be dropped (Line 1 of HandleGoal).

   Constraints on universally quantified variables are not handled in the same way as existential variable constraints. Indeed, care must be taken when matching $\forall$ variables in effects and preconditions. For example, given the goal $\forall person_1$ $\forall person_2$ likes($person_1$, $person_2$), one may be tempted by an effect of the form $\forall p$ likes(p, p). Matching the goal and effect would, of course, produce the constraint $person_1 \approx person_2$. While we could simply say that the match is invalid, since the effect is less general than the goal, The proper way to view this constraint is as a precondition. That is, the goal will only be satisfied if $\forall person_1, person_2$ $(person_1 = person_2)$. This is actually no different from

[2] Note that if a conditional $\forall$ effect were used to satisfy an $\exists$ goal then a non-universally quantified subgoal would be added to $\mathcal{G}$ instead. For example, `compress *` compresses all files in the current directory that are writable. If `compress *` is used to fulfill the goal of compressing all files in the directory, then all the files must all be writable, but if the goal is merely to compress a single file, then it is sufficient for that file alone to be writable.

the existential case – it just happens that in the existential case we can take a shortcut by posting binding constraints to $\mathcal{B}$, whereas here we store the constraints in the newly-generated subgoal. This ensures that neither the goal nor the effect is incorrectly restricted by the variable bindings. The subgoal $\forall person_1, person_2\ (person_1 = person_2)$ can in theory be satisfied by the same mechanisms that apply for other goals – namely, finding an effect that produces it, or finding that the context of the goal entails it. However, since SADL effects don't include binding constraints, the only option is to check whether the binding constraints are redundant when evaluated in the goal context, in which case the effect is not less general than the goal.

2. Alternatively, if the goal is of the form $\Phi \Rightarrow \Psi$, where Φ contains all the universally quantified variables in Ψ, then PUCCINI can subgoal on obtaining LCW on the context Φ. If Φ contains terms labeled with **initially**, these terms are labeled with **initially** in the LCW goal as well. Once PUCCINI has LCW(Φ), the universe of discourse is completely represented in its knowledge base. At this point PUCCINI generates the universal base and subgoals on achieving it (Line 2(a) of Reduce, Figure 5.3). Note that this strategy differs from the classical case since it involves interleaved planning and execution. Given the goal of printing all files initially in `/papers`, PUCCINI would plan and *execute* an `ls -a` command, while protecting the **initially** goal, then, after executing the `ls -a`, would plan to print each file it found, and finally execute that plan.

3. In case neither mechanism alone is sufficient, PUCCINI also considers combinations of these mechanisms to solve a single $\forall$ goal, via a technique called *partitioning*, shown in Line 2(b) of Reduce, Figure 5.3. Partitioning simply consists of splitting the context of the goal into two more restricted contexts whose disjunction entails the original context. For example, the goal of printing all files in a directory could be replaced with the goals of printing the postscript

files and the non-postscript files. Partitioning can be on predicates or variable bindings (*i.e.*, codesignation and non-codesignation constraints). Partitioning on codesignation constraints can be used to overcome the problem mentioned above when the $\forall$ effect is more specific than the goal. We could partition the context into $person_1 \approx person_2$ and $person_1 \not\approx person_2$. The former context would enable us to use $\forall p$ likes(p, p) in partial fulfillment of the goal, and then tackle the more difficult part another way.

It would seem as if partitioning introduces a huge combinatorial disaster, which would be the case if any predicate or binding relation could be used to split the goal, but there's a saner way. Given a $\forall$ effect, like the one above, that partially fulfills the goal, we partition only on the preconditions introduced by that goal. That is, we never partition until *after* we've linked to the $\forall$ goal, and we do so only as a way of eliminating terms from the precondition. Partitioning of a goal that is not to be solved via $\forall$ linking is unnecessary and unhelpful, since LCW reasoning only works on positive conjunctions, and partitioning introduces negation into at least one context. Partitioning on terms that don't appear in the effect preconditions is also unhelpful, since the resulting partitions will be no easier to solve than the original. That's because as far as $\forall$ links are concerned, the context is only useful in eliminating preconditions. We believe this partitioning scheme can solve any goal that partitioning on any condition could solve (but see Section 6.4 for a discussion on completeness).

Note also that the classical universal base mechanism requires that a universe be static and finite. PUCCINI correctly handles dynamic universes, using the threat resolution techniques described in Section 5.3.1. Furthermore, PUCCINI's policy of linking to $\forall$ effects handles infinite universes, but this is not of practical import.

For example, suppose Sloppy Joe has to clean the kitchen: $\forall i$ (**initially**(in(i, `kitchen`)) $\Rightarrow$ **satisfy**(clean(i))). There are only three actions that Joe is willing to

use

1. Loading and running the dishwasher cleans everything small enough to fit in the dishwasher: $\forall d$ (**when**(fits.in (d, `dishwasher`)) **cause**(clean(d)))

2. Spraying an object with a hose will clean that object, provided it's not so small that the stream of water washes it away: (**when**(heavy(o) **cause** (clean(o))))

3. Looking around reveals the objects in the kitchen.

Say Joe first decides on loading the dishwasher. This results in the subgoal: $\forall i$ **satisfy**(fits.in(i, `dishwasher`)). This subgoal can't be achieved, since some objects, such as the stove, are too big to fit in the dishwasher. However, we can partition on fits.in(i, `dishwasher`). Then loading the dishwasher can be used to satisfy one partition, since the context entails the goal. That leaves the remaining goal of cleaning all objects in the kitchen that won't fit in the dishwasher. This goal can be achieved by first looking around to obtain LCW on the objects in the kitchen that don't fit in the dishwasher, and then spraying each one of those objects with the hose, assuming they're heavy enough to stay put under the water pressure; but since they're too big to fit in the dishwasher, let's assume they all are.

5.2.6 LCW *Goals*

As we mentioned, one way of solving universally quantified goals is to first obtain LCW on the universe of discourse. But how is that to be accomplished? What we do is post the desired LCW formula to the goal agenda $\mathcal{G}$, and then re-evaluate the $\forall$ goal once the LCW goal has been achieved (Line 2(a) of Reduce, Figure 5.3). As with standard goals, LCW goals can be satisfied by adding a link from a new or existing action or linking from A_0 (i.e. using LCW information stored in the database $\mathcal{L}$).

LCW goals can also be handled by two other techniques, known as *Intersection Cover* and *Enumeration*. These techniques correspond directly to the Conjunction

and Composition rules described in Chapter 2. Both techniques can be easily understood by thinking of LCW formulas in terms of sets. The conjunction of two LCW formulas, Φ and Ψ can be thought of as representing knowledge about the intersection of the sets described by Φ and Ψ. For example, if Φ is `in.dir(`f`, /bin)` and Ψ is `postscript(`f`)`, then LCW(Φ) represents knowledge of all files in `/bin`, LCW(Ψ) represents knowledge of all postscript files, and LCW($\Phi \wedge \Psi$) represents knowledge of all postscript files in `/bin`, the intersection of the two sets.

One way of identifying all members $\Phi \wedge \Psi$ would be to first identify all members of Φ, and then for each member, determine whether it is also a member of Ψ. We call this approach *Enumeration*, because it relies on enumerating elements of Ψ. A goal LCW($\Phi \wedge \Psi$) can be solved by first obtaining LCW(Φ), and then for each ground instance $\Phi\theta$ of Φ, subgoal on LCW($\Psi\theta$). For example, to find all postscript files in `bin`, one could first find out all files in `bin` and then for each file in that directory, find whether the file is postscript.

It may be costly to enumerate all members of the extension of Φ if that set is large. Or it may be the case that Φ and Ψ have no variables in common, in which case enumeration of Ψ is pointless. Another alternative is to realize that if one knows the members of *both* sets, then one knows their intersection, so a goal LCW($\Phi \wedge \Psi$) can be achieved by subgoaling on LCW(Φ) $\wedge$ LCW(Ψ). For example, one way of finding out all postscript files in `bin` is to find out all files in `bin` and find out all postscript files. We call this approach *Intersection Cover*.

5.2.7 Making Assumptions (Leap before you look)

As discussed earlier, it is often desirable to execute actions for their conditional effects even when the precondition is not known ahead of time to be true. For example, the UNIX command `ls papers` will only find the file `paper.tex` if `paper.tex` is in directory `papers`, but subgoaling to ensure that `paper.tex` is in directory `papers` misses the point. All we really care about is that we will be able to verify that

`paper.tex` is there after the fact. While preconditions explicitly tagged with **satisfy** must always be achieved prior to execution,[3] unannotated SADL preconditions need not be, provided they can be verified later.

What we want is the ability to *assume* the precondition is true, and to later *verify* that the assumption was valid by performing an observation. Formally, an assumption is a quadruple $\langle S_p, p, e, S_e \rangle$, where p is a precondition of some effect of S_p. p is assumed to be true, and is to be verified by effect e of step S_e. Because the observation will only be valid if the condition p remains unperturbed, we must protect p over the interval between S_p and S_e.

The observant reader will notice a striking similarity between assumptions and causal links. In fact, they are identical to causal links, with the exception that the order of the producer and consumer is reversed! Because we want to consider supporting these preconditions by *either* prior observation *or* assumption, we can accomplish this feat quite simply by omitting the ordering constraint that would normally be placed between producer and consumer (See Figure 5.4). Relaxing this constraint allows for self-links as well — in fact, that is the most common type. Consider the following effect of `ls`: ∀ *!f* **when** in.dir(*!f*, *d*) **observe**(in.dir(*!f*, *d*))

We can satisfy the in.dir precondition by linking to the effect of the same action. If the desired file is not in the directory, the assumption will be proven false, and the plan will fail. In the discussion of SADL, self-links were the *only* type of assumptions permitted. The precondition had to be verified immediately after execution. The reason for this is that verifying the precondition later introduces correlations into the agent's knowledge, and the knowledge representation doesn't support correlations. However, the causal link structure of the plan provides a limited mechanism for keeping track of correlations, and thus we can exploit it to provide greater flexibility. This

[3] **satisfy** goals, by definition, must be known to hold at the time they are required. The rational for annotating action preconditions (as opposed to effect preconditions) with **satisfy** is that the effects of the action are undefined if the preconditions aren't true; this introduces an arbitrary about of uncertainty about the resulting state of the world, thus invalidating all of the agent's beliefs.

greater flexibility afforded by assumptions is largely responsible for the simplifications to the UNIX domain discussed in Section 7.1.

Bookkeeping

While we can handle assumptions elegantly by lifting the ordering constraints imposed along with observational links, that doesn't free us from bookkeeping. The effects that are supported by assumptions are still contingent, and we must exercise care in what we do with them. In particular, we should not store them in the agent's knowledge base or execute actions whose primary preconditions are supported by them until they have been observed.

This bookkeeping is really quite simple. Once the link $A_p \stackrel{q,e}{\rightarrow} A_c$ is known to represent an assumption, as opposed to an observational link (*i.e.*, after the A_c has been constrained to come before the A_p) all steps whose primary (**satisfy**) preconditions are supported by A_c are required to follow A_p. When it comes time to execute an action, all effects whose preconditions are unknown, including assumptions, must be asserted as unknown in the agent's knowledge base. Additionally, all effects whose assumed preconditions have been verified should be asserted as true (this is valid, since the causal link guarantees that the condition wasn't changed by the agent in the interim.

5.3 Resolving Threats

We have said that the purpose of a causal link $A_p \stackrel{q}{\rightarrow} A_c$ is to protect condition q from being violated during the interval between the execution of A_p and the execution of A_q. How exactly does PUCCINI prevent condition q from being violated? We could just view the causal link as a constraint, and reject plans that have inconsistent constraints, but that would require doing costly constraint satisfaction for every partial plan the planner considers. We instead take a more proactive approach, by adding

new constraints to the plan whenever a causal link is *potentially* violated, to ensure that it is not violated. See [39] for an excellent discussion of the various tradeoffs involved.

When the condition is potentially violated, but could be saved, this is referred to as a *threat* to the link, and the planner must take evasive action to avoid the threat. The three standard threat avoidance mechanisms are *promotion, demotion,* and *confrontation* (Lines 1-3 of HandleThreats, Figure 5.6). Promotion and demotion order the threatening action before the link's producer or after its consumer, respectively. Confrontation works when the threatening effect is conditional; the link is protected by subgoaling on the negation of the threat's antecedent [72]. For ordinary causal links, PUCCINI uses these standard techniques for resolving threats. But PUCCINI has other kinds of causal links, for which the standard techniques are insufficient.

5.3.1 *Threats to Forall Links*

$\forall$ links can be handled using the same techniques used to resolve ordinary threats: demotion, promotion, and confrontation. Additionally, the following rule applies (Line 5 of HandleThreats, Figure 5.6).

- **Protect forall:** Given a link $A_p \stackrel{E,G}{\rightarrow} A_c$ in which

$$G = \forall x (\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x) \Rightarrow \mathtt{S}(x))$$

  and a threat A_t with effect $\mathtt{P}_1$(foo), subgoal on achieving $\mathtt{S(foo)} \vee \neg \mathtt{P}_2(\mathtt{foo}) \vee \ldots \vee \neg \mathtt{P}_n(\mathtt{foo})$ by the time A_c is executed.

For example, suppose a $\forall$ link recording the condition that all files in `mydir` be group readable is threatened by action A_t, which creates a new file, `new.tex`. This threat can be handled by subgoaling to ensure that `new.tex` is either group readable or not in directory `mydir`.

5.3.2 *Threats to* LCW

The other way to satisfy a $\forall$ goal is to subgoal on obtaining LCW, and then add the universal base to $\mathcal{G}$. However, since LCW goals can also get clobbered by subgoal interactions, PUCCINI has to ensure that actions introduced for sibling goals don't cause the agent to *lose* LCW. For example, given the goal of finding the lengths all files in `/papers`, PUCCINI might execute `ls -la`. But if it then compresses a file in `/papers`, it no longer has LCW on all the lengths.

To avoid these interactions, we use LCW links, which are like standard causal links except that they are labeled with a conjunctive LCW formula. Since LCW(P(x) $\wedge$ Q(x)) asserts knowledge of P and Q over all the members of the set $\{x \mid \mathtt{P}(x) \wedge \mathtt{Q}(x)\}$, an LCW link is threatened when information about a member of the set is possibly lost or a new member, for which the required information may be unknown, is possibly added to the set. These two cases are simply information loss and domain growth, respectively (see Section 3.5). Like threats to ordinary links, threats to LCW links can be handled using demotion, promotion, and confrontation. In addition, threats due to information loss can be resolved with a new technique called *shrinking*, while domain-growth threats can be defused either by shrinking or by a method called *enlarging*.

Information Loss We say that A_t threatens $A_p \xrightarrow{G} A_c$ with information loss if $G =$ LCW($P_1 \wedge \ldots \wedge P_n$), A_t possibly comes between A_p and A_c, and A_t contains an effect that makes R unknown, for some R that unifies with some P_i in G. For example, suppose PUCCINI's plan has a link $A_p \xrightarrow{H} A_c$ in which

$$H = \mathrm{LCW}(\mathtt{in.dir}(f\mathtt{,/papers}) \wedge \mathtt{length}(f, n))$$

indicating that the link is protecting the subgoal of knowing the lengths of all the files in directory `/papers`. If PUCCINI now adds the action `compress myfile.txt`, then the new action threatens the link, since `compress` has the effect of making the

length of `myfile.txt` unknown.

- **Shrinking** LCW**:** Given a link with condition LCW($\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x)$) and threat causing $\mathtt{P}_1$(`foo`) to be unknown (or true), PUCCINI can protect the link by subgoaling to achieve $\neg\mathtt{P}_2(\mathtt{foo}) \vee \ldots \vee \neg\mathtt{P}_n(\mathtt{foo})$[4] at the time that the link's consumer is executed. For example, compressing `myfile.txt` threatens the link $A_p \xrightarrow{H} A_c$ described above, because if `myfile.txt` is in directory `/papers`, then the lengths of *all* the files in `/papers` are no longer known. However, if `in.dir(myfile.txt,/papers)` is false then the threat goes away.

These cases are handled by line 4 of HandleThreats (Figure 5.6).

Domain Growth We say that A_t threatens $A_p \xrightarrow{G} A_c$ with domain growth if $G =$ LCW($P_1 \wedge \ldots \wedge P_n$), A_t possibly comes between A_p and A_c, and A_t contains an effect that makes R true, for some R that unifies with some P_i. For the example above in which the link $A_p \xrightarrow{H} A_c$ protects LCW on the length of every file in `/papers`, addition of an action which moved a new file into `/papers` would result in a domain-growth threat, since the agent might not know the length of the new file. Such threats can be resolved by the following.

- **Shrinking** LCW (described above): If PUCCINI has LCW on the lengths of all postscript files in `mydir`, then moving a file into `mydir` threatens LCW. However, if the file isn't a postscript file, LCW is not lost.

- **Enlarging** LCW**:** Given a link with condition LCW($\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x)$) and threat causing $\mathtt{P}_1$(foo) to be true, PUCCINI can protect the link by subgoaling to achieve LCW($\mathtt{P}_2(\mathtt{foo}) \wedge \ldots \wedge \mathtt{P}_n(\mathtt{foo})$) at the time that the link's consumer is executed. For example, moving a new file `xii.tex` into directory `/papers`

[4] Note the difference between shrinking and protecting a ∀ link. Unlike the ∀ link case, shrinking does not have a disjunct corresponding to `S(foo)`.

threatens the link $A_p \xrightarrow{H} A_c$ described above, because the length of `xii.tex` may be unknown. The threat can be resolved by observing the length of `xii.tex`.

Note that an effect which makes some P_i *false* does *not* pose a threat to the link! This corresponds to an action that moves a file *out* of `/papers` — it's not a problem because one still knows the lengths of all the files that remain.

HandleThreats($\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{G}$) $\forall \ell \in \mathcal{C}$ threatened by some step S_t either

1. Promote: Add $S_c \prec S_t$ to $\mathcal{O}$.
2. Demote: Add $S_t \prec S_p$ to $\mathcal{O}$.
3. Confront: If threatening effect has a precondition, add its negation to $\mathcal{G}$.
4. If LCW goal choose one of
   (a) Enlarge LCW. If condition is $\text{LCW}(\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x))$ and threat is **cause**($\mathtt{P}_1$(foo, T)) add $\langle \text{LCW}(\mathtt{P}_2(\mathtt{foo}) \wedge \ldots \wedge \mathtt{P}_n(\mathtt{foo})), S_c \rangle$ to $\mathcal{G}$.
   (b) Shrink LCW. If condition is $\text{LCW}(\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x))$ and threat is **cause**($\mathtt{P}_1(\mathtt{foo}, \mathtt{U} \vee \mathtt{T})$ add $\langle \neg \mathtt{P}_2(\mathtt{foo}) \vee \ldots \vee \neg \mathtt{P}_n(\mathtt{foo}), S_c \rangle$ to $\mathcal{G}$.
5. If $\forall$ goal
   (a) If condition is $\mathtt{P}_1(x) \wedge \mathtt{P}_2(x) \wedge \ldots \wedge \mathtt{P}_n(x) \Rightarrow \mathtt{S}(x)$ and threat is $\mathtt{P}_1$(foo), add subgoal $\langle \mathtt{S(foo)} \vee \neg \mathtt{P}_2(\mathtt{foo}) \vee \ldots \vee \neg \mathtt{P}_n(\mathtt{foo}), S_c \rangle$.
6. **hands-off** goal
   (a) If threatening effect is of the form **cause**(φ), add subgoal of the form $\langle \textbf{initially}(\varphi), S_t \rangle$.

Figure 5.6: Procedure HandleThreats

5.3.3 *Threats to* **hands-off**

As mentioned in Section 3.4.2, **hands-off**(φ) is not violated by an action with an effect **cause**(φ, `T`) if **initially**(φ, `T`) can be established. Similarly for **cause**(φ, `F`). Thus one way of protecting a threat to **hands-off**(φ) is to subgoal on **initially**(φ). This is handled by line 6 of HandleThreats (Figure 5.6).

IsExecutable?(A_E)

Return true iff all of the following hold:

- $\mathcal{E}(A_E) =$ `unexecuted`
- All preconditions of A_E are satisfied
- All actions necessarily before A_E have been executed
- No pending action has an effect that clobbers any effect of A_E.
- A_E does not threaten any links
- All parameters are bound.

Figure 5.7: Function IsExecutable

5.4 *Controlling Execution*

As with other decisions made by the planner, we describe execution in terms of flaws and fixes. For a plan to be complete, all actions must be executed. Thus an unexecuted action is a flaw in the plan, whose fix is to execute it. An action should not be executed until certain minimal criteria are met. These criteria are tested in the function IsExecutable? (Figure 5.7).

- All its preconditions must be satisfied, since otherwise the effects of the action are undefined.

HandleExecution($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{E}$)
if $A_E \in \mathcal{A}$ and IsExecutable?(A_E):

- Add $A_l ; A_E$ to $\mathcal{O}$, where A_l is the most recently executed action in the plan, or A_0, if no actions have been executed.
- Call the exection procedure specified in the `interface` of the action (see Appendix B for examples).
- Set $\mathcal{E}(A_E) = \texttt{executed}$
- Update the model
  1. Effects of A_E whose preconditions were known true or verified by observational effects of A_E are asserted to be true.
  2. Effects of previous actions whose assumed preconditions were verified by this action are asserted to be true.
  3. Effects of A_E whose preconditions are unknown (including assumptions) are asserted to be unknown.
- Update $\mathcal{B}$. If there were **observe** effects, add all ground instances, returned by the sensor function, to the plan. If any $\forall$ effects were used to support $\exists$ goals, nondeterministically choose one way of satisfying the goal with the effects.
- If execution fails or bindings inconsistent, then return failure.

Figure 5.8: Procedure HandleExecution

- It must be consistent for the action to be ordered before all other unexecuted actions in the plan, since any action ordered prior to it must be executed prior to it.

- The action must not be involved in any threats, since executing the action fixes its order in the plan, thus eliminating from consideration threat resolution techniques that involve action orderings.

- The physical action executed must be unambiguous. In particular, all plan-time variables used as parameters to the command the agent is to execute must be bound before the action can be executed. For example, a command such as "Put block A on block x", where x is a variable, is obviously ambiguous, as is `ls` d.

- Finally, it must be possible to verify the effects of the action that are used to support causal links. Note that, in the case of assumptions, it is not necessary that they be verified before the action is executed, but there must be a plan to verify them at some point before they are actually needed.

The action is executed by invoking a procedure (represented using `exec-func` and `translation` in Appendix B. For example, in `ls`, the procedure consists of sending the string `"ls -F"` to a UNIX shell). If there is a sensor function (`sense-func`) associated with the action, then any output returned from the execution procedure is processed by the sensor function, which returns sensed information in the form of bindings to run-time variables. After the action is physically executed, the knowledge base is updated with the effects of the action, including any sensed information returned via variable bindings. While these updates usually consist of adding information to the knowledge base, they could involve removing it as well. For example, effects can have truth values of `U`, indicating that the literal in question will change in

some undetected manner, becoming unknown to the agent. E.g, executing `compress bigfile` changes the size of `bigfile` without notifying us (or the Softbot) what the new size is. Thus the size becomes unknown. Another example is when a conditional effect like `chmod u+w *` (which changes the permission of all files, owned by the user, in a given directory) is executed, but the precondition is unknown.

In UCPOP, the choice of what flaw to work on is *not* a backtrack point; goal ordering and threat ordering influence efficiency but not completeness. This is not the case for execution. Because executing an action entails fixing its position in the plan, execution can restrict the number of subsequent choices available. For this reason, execution is different from other flaws in that, for completeness, the decision to execute an action must be a nondeterministic choice — that is, it must be regarded by the planner as a choice point, to backtrack over as necessary.[5] In the worst case, this means that the planner will consider all possible orderings of actions, which is factorial in the number of actions. There is a tradeoff here between completeness and efficiency. In the absence of irreversible actions, the decision to execute rarely affects completeness, and we have found it acceptable to treat execution the way we treat other flaws, despite the theoretical loss of completeness.

5.4.1 Impact of Execution

We described execution as a planner choice point, but as the previous discussion illustrates, there are clearly ways in which execution differs from choices such goal establishment and threat resolution. We now discuss what it means to execute an action in a partial plan, both at an operational level, and how it affects the nature of the search space. We then talk about potential problems that execution raises, and ways those problems can be circumvented.

---

[5] An alternative to this approach (discussed in the context of combining forward-chaining and plan-space planning), which also preserves completeness, but at the expense of a large (in our case infinite) branching factor is discussed in [40].

Execution of an action involves changes to the world, to the agent's knowledge of the world, to the current plan, and to the entire search space. The most obvious change is to the state of the world. Hopefully, whatever causal effects are listed in the action description are now true in the world, and everything else remains as it was. That may not be the case: Execution may simply fail. In general, this is due to exogenous events that make a precondition false, or to unsatisfied preconditions not modeled by the domain theory. Examples of such preconditions are that the file server must be running for `ls` to succeed, and a robot's arm must not be broken if it is to pick up a tool.[6] In any event, if the desired effects have not been achieved, it is necessary to backtrack. Assuming the action was successfully executed, the agent must process whatever information is returned by its sensors and record the effects of the action in its knowledge base.

In addition to changing the world and its knowledge of the world, PUCCINI must change the current plan, by marking the action as executed and adding new binding and ordering constraints. The run-time variables in the effects must be bound to the values determined by the agent's sensors. In the case of observational $\forall$ effects, the number of binding constraints may be arbitrary. Since the ordering constraints imposed on actions dictate relative order of *execution*, once an action A_{exec} has been executed, we add a constraint, $A_{prev}\,;A_{exec}$, where A_{prev} is the previous executed action (or A_0 if no other actions have been executed). This ensures that all unexecuted actions, including actions not yet added to the plan, will follow A_{exec}. If executing the action reveals the values of a $\forall$ variable, then even more profound changes to the plan may be required. For example, say the plan contained the $\exists$ goal of finding some file in directory `papers` and printing it, where finding a file is to be achieved by `ls`. Once `ls papers` is executed, the agent discovers there are five files there. It now

[6] Modeling such preconditions is pointless, since they are outside the ability of the agent to either control or reliably detect. However, they bring home the point that the qualification problem is alive and well in software domains.

has a choice of printing one of those five files, requiring a planner choice point. Thus five new plans are generated, one for each file in `papers`, and PUCCINI chooses one of them. Note that if the goal had been to print *all* of the files, there would be only one plan, with five new open goals, one for each of the files that must be printed. These five new goals constitute the universal base of the original ∀ goal (see Section 5.2.2).

In addition to the above changes to the world state, knowledge base, and the current plan, there may be a changes to the search space itself. Once an action has been executed, other plans in the search space will no longer be consistent with the state of the world and, to cope with this discrepancy, some extra work may be required if these plans are ever adopted. We refer to adopting such a plan as *backtracking over execution* and discuss it in length below.

5.4.2 Backtracking Over Execution

Because plans are sometimes partially executed before they are fully worked out, it may happen that actions are executed in the course of following a blind alley, and that these actions need to be *undone* somehow to get back to the true path to the goal. We call this undoing of executed actions *physical backtracking*

If we want to support all possible search strategies, even BFS, it is necessary to be able to jump to any plan in the fringe of the search tree, regardless of where it is relative to the current plan. This may require not just backtracking, but *forward*-tracking as well: re-executing actions that were previously executed but later undone. Conceptually, going from plan $\mathbf{p}_a$ to plan $\mathbf{p}_b$ entails backtracking from $\mathbf{p}_a$ to the common ancestor of $\mathbf{p}_a$ and $\mathbf{p}_b$ and then *forward*-tracking to $\mathbf{p}_b$.

A few things are important to note. First, when we interleave planning and execution, we are no longer just searching through plan-space. We are simultaneously searching through the state-space of the world and plan-space. However, since we have incomplete information about the current state, we never know exactly what state the world is in. Also, while the state of the world is changing, our incomplete information

about the world state is changing in response to the causal and observational effects we execute. So while searching through world-state space, we are also searching through the space of knowledge about the world. It should be clear by this point that reliably returning the world to the exact state it was in at some earlier time is, in the general case, impossible. Fortunately, there are backtracking strategies that work in all but the most unforgiving domains. Before discussing these strategies, we present a few useful definitions.

We want to capture the notion of returning the world back to the same state it was in previously, but by the Unique Names Assumption, returning to the same situation is impossible; the history is, in effect, part of the state. Furthermore, returning to the same state of knowledge is generally impossible as well, since most agents can't forget on demand. So we want some notion of two situations being equivalent, with the possible exception of their history and the agent's knowledge. We do so simply by exempting the DO and K predicates from consideration:

Definition 5.1 (equivalence) *Two situations s, s' are said to be* equivalent ($s \equiv s'$) *if for all fluents φ, not including K and DO, $\varphi(s) \Leftrightarrow \varphi(s')$*

Given this definition, we can say what it means to "undo" an action:

Definition 5.2 (inverse) *An action sequence $\{a\}_1^{n-1}$ is an (unconditional) inverse of action a iff for any states s, s', $(DO(a,\ s) \equiv s') \Rightarrow (DO(\{a\}_1^{n-1},\ s') \equiv s)$.*

It is not in general the case that if $\{a\}_1^{n-1}$ is the inverse of a, then a is the inverse of $\{a\}_1^{n-1}$. For example, `rm backup` is the inverse of `cp important.file backup`, but `rm backup` itself has no inverse (as many frustrated UNIX users have discovered).

Another important point is that any action a_{obs} that has only observational effects does not affect predicates other than K and DO (DO(a_{obs}, s) $\equiv s$). Therefore all such purely observational actions have an inverse, namely the null action a_{nop}. This has important implications to physical backtracking, since we never need to worry

about backtracking over observational actions. A possible concern might be that while we can trivially return the world to its original state, we may not be able to return the agent's knowledge about the world to its previous value. While that is true, it is unimportant, since executing a purely observational action only increases the agent's knowledge about the world, and having more knowledge about the world cannot make a previously attainable goal unachievable.

While some actions that change the world, such as `pushd /bin` have an inverse (namely `popd`), not many do. Even an action such as `cd /bin`, which can clearly be undone, does not have an inverse, by the above definition, since the precise command needed to undo the `cd` depends on what the current directory was when the `cd` was executed. It is often the case that a given action sequence can reverse the effects of an action in some states, but not in all. For example, the action of deleting a file is reversible just in case there is a backup of the file. If no backup was made before deleting the file, then the file is irretrievably lost. Otherwise, reversing the action is merely a matter of restoring from the backup.

Definition 5.3 (conditional inverse) *We say an action $\{a\}_1^{n-1}$ is a conditional inverse of action sequence a when there is some condition ρ such that for any situations s and s', if $\rho(s)$ and $DO(a,\ s) \equiv s'$ then $DO(\{a\}_1^{n-1},\ s') \equiv s$.*

In general, an action may have many conditional inverses, each with its own precondition ρ_i. In this case, all the inverses, taken together, form a contingent plan for reversing the action. This contingent plan is applicable in all situations in which $\rho_1 \vee \rho_2 \vee \ldots \vee \rho_n$ is true. If this disjunction is true in all situations, then there is an appropriate inverse of the action for each situation. In this case, we say that the action is *reversible.*

Definition 5.4 (reversible) *An action a is reversible iff for any world states s, s', $(DO(a,\ s) \equiv s') \Rightarrow \exists\ \{a\}_1^n$ such that $DO(\{a\}_1^n,\ s') \equiv s$.*

If an action a is not reversible, but has a conditional inverse applicable in some situations, we say a is *conditionally reversible*. Otherwise, we say it is *irreversible*.

5.4.3 Policies for Backtracking Over Execution

Pursuing any policy of backtracking over execution presupposes that it's okay to make a mistake. If solving a goal requires the agent to execute irreversible actions, and if the consequences of those actions are potentially dire, then it may be more appropriate to use a contingency planner, *i.e.*, to plan for every possible contingency in order to ensure that the agent can't paint itself into a corner. Of course, contingency planning is not always feasible, and a contingency plan doesn't always exist; an agent trying to cross a mine field can spend all day thinking, but unless it knows where the mines are, that won't do it much good.

Assuming no actions are irreversible, or at least that mistakes aren't deadly, there are two strategies that PUCCINI supports for backtracking over execution: "Safety first" and "Lazy"

Safety-first backtracking

If all actions are conditionally reversible, but not all actions are unconditionally reversible, then the agent can guarantee that it can get out of any situation it gets into by always ensuring that that the undo preconditions ρ of actions are satisfied before the actions are executed, and then executing the appropriate inverses when backtracking. This is the equivalent of tying a rope to itself before attempting to climb down into a pit. However, if some condition ρ is unachievable, or inconsistent with the goal, then this strategy will cause the planner to fail when it might otherwise have succeeded. If, on the other hand, the agent should not choose to ensure that ρ is true, then it runs the risk of being stuck in a pit, which could also lead to failure. At least in the case of achieving ρ, when the agent's plan fails, the agent is

in a recoverable state.

We call this policy “safety first,” since it puts recoverability ahead of completeness. This policy may also yield inefficiency, since inferring the conditions ρ and ensuring that they are satisfied could involve considerable computational cost. We can avoid the cost of determining what the undo conditions are by requiring they be stated explicitly in the action descriptions. However, that still leaves the problem of achieving them.

We require all actions to be labeled with an undo precondition, along with a procedure for undoing the effects of the action. The undo precondition could be nil, in which case the action is irreversible (or t, in which case the action has an unconditional inverse). Safety-first requires that irreversible actions be avoided.

Lazy backtracking

The safety-first policy is neither complete nor efficient. Furthermore, if all actions are reversible, then it is still possible to backtrack over execution without adding an extra source of incompleteness, unless there are **initially** goals that have not yet been achieved. The intuition is as follows. If we don't know what the state of the world was when a given plan was created, then we may not be able to find the correct sequence of actions to return to that state, and some information about that state will be irretrievably lost (hence some **initially** goals may become unachievable). However, if all we care about is reaching a state that satisfies some goal condition, then all that matters is whether such a state is reachable. Because all actions are reversible, any state that was reachable before execution will be reachable afterward. It is irrelevant that the planner may be unable to return back to the original state, since the goal condition does not depend on going through that particular state. Merely relying on the reachability of past states is sufficient. But how is the planner to backtrack over execution? Since the undo conditions are not necessarily satisfied, it cannot simply execute the inverses of the executed actions. It must actually plan in order

to restore the state. It turns out that it is unnecessary to backtrack to the original state; it is sufficient to re-establish the invalidated conditions of the plan by using plan repair to retract the causal links and associated constraints supporting those conditions and replanning to achieve them. In addition to constraints, the planner records modifications to the state of the world that are not already reflected in the plan structure. These are regarded as effects of a new "dummy step" representing the current state.

For example, say the planner is backtracking from plan $\mathbf{p}_x$ to $\mathbf{p}_y$, and $\mathbf{p}_x$ involved the execution of `cd newdir`, but when $\mathbf{p}_y$ was created, the current directory was `olddir`. Unless $\mathbf{p}_y$ had a commitment to being in `olddir`, doing another `cd` is unnecessary. It may be the case that subsequent refinements to $\mathbf{p}_y$ will result in a commitment to be in `olddir`, `newdir`, or some other directory, but those can be dealt with as the need arises.

Note that this approach also can result in considerable computational cost, because the cost of reestablishing the invalidated conditions can be arbitrarily high. So either approach, safety-first or lazy, comes with an extra planning cost, either to satisfy undo conditions or to reestablish invalidated conditions.

Another concern is that the planner might spend all its time re-establishing invalidated conditions, and never actually make progress along any plan branch. This would happen if it decided to backtrack while re-establishing conditions from the last backtrack point, effectively making backwards progress, and reconsidering the same (or equivalent) plans repeatedly. Such behavior would be possible with a sufficiently bad search strategy, but simply assigning a high penalty to plans requiring physical backtracking ensures that the planner will make progress along one physical branch before switching to another, thus always pushing the fringe of the search space forward.

Retry, Replan, Fail?

One might imagine buying out of the physical backtracking problem entirely, by simply replanning from scratch whenever such backtracking would be required. This is the approach taken by IPEM [1]. It has the advantage that backtracking is simpler, but the disadvantage of doing considerably more search. If changes to the world are involved, it is also vulnerable to the kind of looping described above, and worse. By throwing away the state associated with the planner's search space, the agent has no knowledge of dead ends, so there is no way to avoid making the same mistakes repeatedly.

5.5 Computational Complexity

STRIPS planning is PSPACE-hard [6], so like simpler planners, PUCCINI takes time that is exponential in the size of the problem. However, the added expressiveness and flexibility over other planners does not come at an unreasonable price; in fact, the computational complexity of evaluating individual plans is essentially the same as it is for UCPOP [72]. The only caveat is that the complexity of LCW reasoning depends on the actual domain theory, and could be quite large. As we will see, LCW reasoning dominates the time complexity of plan evaluation. Below we consider the costs of the PUCCINI extensions.

- PUCCINI uses LCW both to prune away redundant sensing and to solve universally quantified goals. LCW reasoning is polynomial in the size of the agent's knowledge, and thus is polynomial in the size of the input to the planner. As we show in Section 7.2, this polynomial growth doesn't materialize in practice, but consulting $\mathcal{M}$ and $\mathcal{L}$ does add a significant extra per-plan cost in practice. However, as we demonstrate in Section 7.2, utilizing LCW speeds up planning dramatically by allowing the planner to prune redundant sensing operations and thus visit fewer plans. Planners like UCPOP incur at most cubic-time costs

for plan evaluation, so LCW reasoning does potentially increase the per-plan complexity.

- Executing actions is a constant-time operation, though the constant is so large that it tends to dominate the total running-time of the planner. Additionally, two operations related to execution affect the complexity of plan evaluation.

  1. Detecting whether an action can be executed requires iterating through the steps in the plan to find those that can be safely ordered before all other steps in the plan, which is a cubic-time operation. Fortunately, all the hard work is already done by the planner's test for inconsistent constraints, so the added cost of finding executable actions is negligible.
  2. Updating the agent's knowledge base, including $\mathcal{L}$, after execution is polynomial in the size of the knowledge base.

- The new ways of satisfying open conditions, including **initially** links, $\forall$ links and assumptions, utilize the same algorithm that standard causal links use, and have the same cost, which is dominated by a cubic-time search for inconsistent constraints. As we mentioned, subgoaling on LCW is dominated by the polynomial-time LCW inference procedure. Partitioning $\forall$ goals doesn't contribute to the per-plan cost, but potentially contributes dramatically to the branching factor.

- The new ways of resolving threats don't add to the per-plan cost, with the exception of LCW threats, where additional polynomial-time LCW queries are involved. However, the increased number of refinements adds to the branching factor.

- Backtracking over execution using the lazy method requires checking for invalidated links. The number of links is potentially quadratic in the number of

steps. The condition of each link must be checked against the knowledge base, a linear-time operation.

In practice, PUCCINI is fast enough for simple planning problems, but needs to be more efficient if it is to solve difficult planning problems in real time (which is what we want for softbot applications). As we mention in our discussion of the performance results in Section 7.1, we consider the ultimate bottleneck to be execution time.

5.6 Summary

In this chapter, we presented the PUCCINI planning algorithm, which uses the SADL action language and the LCW knowledge representation language to construct and execute plans in the presence of incomplete information. Novel contributions include solving universally quantified goals in the presence of incomplete information, solving **initially** and LCW goals, using links to record assumptions, and new methods for resolving threats to causal links labeled with LCW, $\forall$ or **hands-off**. We showed that, with the exception of LCW reasoning, the PUCCINI extensions do not affect the computational complexity of plan evaluation.

In the next chapter, we prove that PUCCINI is sound, though not complete.

Chapter 6

FORMAL PROPERTIES

Since a planner is at heart a search program, searching for the solution to a planning problem among the set of all potential solutions, some natural questions to ask about the program are: does it return only correct solutions, and does it return all correct solutions? We refer to property of returning only correct solutions as *soundness*. We refer to the property of returning all correct solutions as *completeness*.[1]

Given that PUCCINI not only searches the space of plans, but also executes those plans, thus changing the state of the world, we must modify the standard definition of soundness to take into account the behavior of PUCCINI. Soundness means that if PUCCINI terminates and reports success, then the goal is achieved. We will show that PUCCINI is sound, but incomplete. Before proceeding directly to the statement of soundness, we introduce some useful definitions and theorems. The proofs for all the theorems are in Appendix A.3.

6.1 Causality Theorems

In his thesis [69], Pednault showed that given the standard assumptions of planning, such as no exogenous changes, and plans of only finite length, any condition φ true at a given point in time must have either been true all along, or must have become true as the result of executing a particular action. This is stated in his Causality theorem, which has the following Corollary (Corollary 3.2 in Pednault's thesis)

[1] As we discuss in Section 6.4, it is not entirely clear what completeness should mean when planning with incomplete information.

Theorem 6.1 (Pednault) *Let $\{a\}_1^n$ be a sequence of actions that is executable in situation s. A condition φ will be achieved by action sequence $\{a\}_1^n$ iff one of the following holds*

1. *There is some prefix of $\{a\}_1^n$, $\{a\}_1^m$ such that*
   (a) $\alpha \not\models R_{\{a\}_1^{m-1}}(\varphi)$ *and*
   (b) $\alpha \models R_{\{a\}_1^{i}}(\varphi)$ *for all $m \leq i \leq n$.*
2. $\alpha \models R_{\{a\}_1^{i}}(\varphi)$, *for all $i = 1 \ldots n-1$.*

Since we make the same assumptions Pednault does, this theorem is valid for us as well. Since SADL goals include knowledge goals (**satisfy** and **initially**) and maintenance goals (**hands-off**), we extend Pednault's theorem to indicate when goals of these varieties are achieved. We do so simply by applying the definitions of regression for each type of goal. We start with **satisfy** goals.

Corollary 6.2 (Causality Theorem for satisfy) *A condition* **satisfy**(φ, tv) *will be true after executing $\{a\}_1^n$ from an initial state axiomatized by α iff one of the following holds*

1. (a) $\alpha \models R_{\{a\}_1^{m-1}}(\Sigma^{a_m}_{\mathrm{KNOW}(\varphi)})$
   (b) *For all $i = m \ldots n-1$, $\alpha \models R_{\{a\}_1^{i}}(\mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}}))$.*
2. $\alpha \models \mathrm{KNOW}(\varphi)$, *and for all $i = 1 \ldots n-1$, $\alpha \models R_{\{a\}_1^{i}}(\mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}}))$.*

Note that the causality theorem for **satisfy** is *not* the same as Pednault's causality theorem, because **satisfy** specifically requires that the condition be known to the agent.

Before proceeding to the causality theorem for **initially**, we introduce some useful lemmas. First, note that a condition of the form **initially**(φ), once true will never become false.

Lemma 6.3 *If* **initially**(φ) *is true in state* $DO(\{a\}_1^m, s)$*, it is true in all states* $DO(\{a\}_1^{m+i}, s)$*, for* $i \geq 1$

Another important observation is that a goal **hands-off**(φ) will always be achieved if every step in the plan preserves the truth value of the φ.

Lemma 6.4 *If* $R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\neg\varphi}^{a_{i+1}}) \wedge \mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}}))$ *for all* $0 < i < n$*, then*

$$R_{\{a\}_1^n}(\textbf{hands-off}(\varphi))$$

Note that the converse is false. An agent can execute an action that affects φ without violating the **hands-off**(φ) as long as the agent knows that φ is not changing truth value, *i.e.*, that its new truth value is the same as its original truth value. However, if the agent didn't know the original truth value, then it would in fact need to avoid affecting φ:

Lemma 6.5 *If* $R_{\{a\}_1^n}(\textbf{hands-off}(\varphi))$ *and* $\neg R_{\{a\}_1^n}(\textbf{initially}(\varphi))$ *then*

$$R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\neg\varphi}^{a_{i+1}}) \wedge \mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}})) \textit{ for all } 0 < i < n$$

We now present the causality theorem for **initially**. Note that **initially** differs from **satisfy** in that the **initially** goal must preserve the condition *before* it is observed, but not afterward. That is, the **initially** goal is achieved when the condition φ is observed, provided that φ has the same truth value it had in the initial state. Once the **initially** goal is achieved, it is true ever after, so there's no need to protect it.

Corollary 6.6 (Causality Theorem for initially) *A condition* **initially**(φ) *will be true after executing* $\{a\}_1^n$ *from an initial state axiomatized by* α *iff one of the following holds*

1. (a) *There is some prefix of* $\{a\}_1^n$, $\{a\}_1^m$*, such that* $\alpha \models R_{\{a\}_1^{m-1}}(\kappa_{\varphi}^{tv}(a_m) \wedge \varphi)$ *and*

(b) *For all* $i = 0 \ldots m-1$ $\alpha \models R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}}) \wedge \mathrm{KNOW}(\Pi_{\neg\varphi}^{a_{i+1}}))$

2. $\alpha \models \mathrm{KNOW}(\varphi)$,

The causality theorem for **hands-off** allows the agent to preserve the truth value of φ by either not affecting the truth value of φ, or by ensuring that any actions affecting φ only result in its having the same truth value. Either way, the truth value of φ hasn't changed.

Corollary 6.7 (Causality Theorem for hands-off) *A condition* **hands-off**(φ) *will be true after executing* $\{a\}_1^n$ *from an initial state axiomatized by* α *iff one of the following holds*

1. *For all* $i = 1 \ldots n-1$, $\alpha \models R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}} \wedge \Pi_{\neg\varphi}^{a_{i+1}}))$.

2. *Either*

   (a) $\alpha \models R_{\{a\}_1^n}(\textbf{initially}(\varphi))$ *and for all* $i = 1 \ldots n-1$, $\alpha \models R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\varphi}^{a_{i+1}}))$
   OR

   (b) $\alpha \models R_{\{a\}_1^n}(\textbf{initially}(\neg\varphi))$ *and for all* $i = 1 \ldots n-1$, $\alpha \models R_{\{a\}_1^i}(\mathrm{KNOW}(\Pi_{\neg\varphi}^{a_{i+1}}))$

6.2 Resolving flaws

The PUCCINI algorithm relies on several subroutines for incrementally building plans, discussed it the previous chapter. So before going on to the soundness proof, we will prove that these subroutines are correct. PUCCINI works by removing flaws from an agenda, and adding fixes to the plan, to eliminate the flaws. To prove that these subroutines are correct, we will show that they do in fact fix the flaws.

HandleGoals removes goals from the agenda, and adds structures to the plan to ensure that the goal is achieved. We rely on the causality theorem to show that HandleGoals is correct. That is, any condition removed from the agenda will be replaced by other conditions that satisfy the causality theorem for the original condition.

Theorem 6.8 (Correctness of HandleGoals) *Given an open condition c, HandleGoals will add to the plan actions or constraints that, if obeyed, will result in c becoming true.*

The causality theorem not only requires the condition to become true; it also requires the condition to remain true until it is needed. HandleGoals is insufficient to ensure this, since new structures added to the plan could result in the condition becoming false. HandleThreats deals with this case by adding new structures to the plan that ensure that the condition is not violated. The proof that HandleThreats is correct also relies on the causality theorem.

Theorem 6.9 (Correctness of HandleThreats) *Given a threatened condition c, HandleThreats will detect the threat and will add to the plan constraints that, if obeyed, will prevent c from becoming violated.*

6.3 Soundness

Below is a brief outline of the soundness proof. The details can be found in the proofs for the individual lemmas, which are in Appendix A.3. We proceed by a standard inductive argument, showing that a loop invariant holds when the planner starts, and that it also holds after each iteration of the planner.

Definition 6.10 ***(The*** PUCCINI ***loop invariant)*** *If plan P satisfies all the subgoals on $\mathcal{G}$, and all assumption links in $\mathcal{C}$ supported by steps that have not yet been executed are labeled with conditions that are true, then executing the remaining (unexecuted) actions in P will result in a situation s_n that satisfies $\mathcal{G}$.*

Since we are interested in building plans incrementally, we will rely on Pednault's Expansion Theorem, which describes the conditions under which executing a plan P', which is a refinement of a plan P, will result in a condition φ being true. φ will be true after executing P' if and only if one of the following conditions holds.

- Some new action (in P' but not in P) makes φ true, and all following actions preserve its truth.
- Some old action (in P' and P) makes φ true, and all following actions preserve its truth.
- φ is true originally (in situation s_0) and all actions in P' preserve its truth.

Thus we can build a plan to achieve some condition Γ as follows. Add Γ to a goal agenda $\mathcal{G}$, then for each subgoal on $\mathcal{G}$, either add a new action that makes that subgoal true, or add conditions to ensure that some existing action makes that subgoal true, while ensuring that we don't violate the conditions that preserve previously solved subgoals.

The inductive argument that PUCCINI is sound begins by showing that the loop invariant holds when PUCCINI is first invoked.

Lemma 6.11 *The* PUCCINI *loop invariant holds on the initial call to* PUCCINI.

We then show that the loop invariant is preserved by the recursive call, regardless of the control path that's taken. This relies on the theorems that HandleGoals and HandleThreats are correct, and together satisfy the Causality Theorem, and that HandleExecution detects when assumptions are invalidated.

Lemma 6.12 *If the* PUCCINI *loop invariant holds before an iteration of* PUCCINI, *it will hold afterward.*

Finally, we show that the loop invariant is true when PUCCINI terminates with success.

Lemma 6.13 *If the* PUCCINI *loop invariant holds before an iteration of* PUCCINI, *and* PUCCINI *halts, returning success, then the loop invariant still holds.*

The above lemmas would seem to capture all that we need to easily prove that PUCCINI is sound, but there's still the matter of backtracking over execution, which is not captured by the nondeterministic algorithm we presented. Fortunately, safety-first backtracking also preserves the loop invariant, and we conjecture that lazy backtracking does as well. It is easy to see that safety-first backtracking preserves the loop invariant, since by executing undo actions, the agent is returning to a $\langle plan, state \rangle$ pair that satisfied the loop invariant, with the only difference being the agent may know more about the state of the world. It is easy to see that increasing the agent's knowledge without changing the plan or the world preserves the loop invariant.

It is less obvious that lazy backtracking satisfies the loop invariant, since it returns to a $\langle plan, state \rangle$ pair that *violates* the loop invariant, and attempts to reestablish the loop invariant by means of plan repair. But the intuition is that constraints in the plan that violate the loop invariant are removed, leaving a plan that once again obeys the loop invariant.

Given that the loop invariant is true in all calls to PUCCINI, and at termination, it is easy to prove that PUCCINI is sound.

Theorem 6.14 (Soundness of PUCCINI) *If* PUCCINI *halts and reports success, then the goal is satisfied.*

6.4 Incompleteness

PUCCINI is based on UCPOP, which is both sound and complete. However, although PUCCINI is sound, it is not complete. The traditional definition of completeness requires a planner to produce a plan that achieves the goal, assuming such a plan exists. Such a definition is too strict when planning with incomplete information. Consider, for example, the following variant of the bomb-in-the-toilet problem. There are two packages, one of which contains a bomb. The agent has time to dunk exactly one of the packages in the toilet. Dunking the package that contains the bomb will

defuse the bomb. Dunking the other package will have no effect, and in the meantime, the bomb will go off. There is obviously a plan that will achieve the goal of defusing the bomb, namely, dunking the correct package. But since the agent doesn't know which package contains the bomb, it has no way of knowing *a priori* which package to dunk. Thus by the traditional definition of completeness, no planner can be complete in such domains.

Even with a more liberal definition of completeness, PUCCINI is incomplete. There are several sources of incompleteness added by the PUCCINI algorithm.

- LCW reasoning is incomplete. Since PUCCINI depends on LCW to solve universally quantified goals, it may fail to solve a goal for which a solution exists due to a failure to infer LCW.

- Execution commits to action ordering prematurely. By ordering the executed action before all other actions, PUCCINI excludes many valid action orders (and hence many valid plans). Completeness would require PUCCINI to consider all possible action orders, which it does not.

- PUCCINI is incomplete when executing plans that contain irreversible actions. As discussed in Section 5.4, only contingency planning can preserve completeness in the presence of irreversible actions.

- The flexibility provided $\forall$ goals, combined with the reality of incomplete information, makes it possible to devise goals that are in principle solvable, but which PUCCINI won't be able to solve. This is only an issue for goals that can't be solved completely by subgoaling on LCW. For example, one could write a $\forall$ goal that was a tautology. Since PUCCINI lacks general theorem-proving capabilities, it would not be able to detect this fact. In the absence of being able to compute the universal base and discover that the goal was trivially true,

PUCCINI would be at the mercy of its ability to find $\forall$ effects that would satisfy the goal.

Chapter 7

EMPIRICAL EVALUATION

7.1 PUCCINI

We demonstrate the effectiveness of the SADL language and the PUCCINI implementation of SADL by showing that many UNIX commands and goals can be represented cleanly and planned with easily. In Section 7.1.1, we show the result of running the planner on ten representative goals from the Softbot domain. In Section 7.1.2, we show a detailed plan trace of one of those goals. In Section 7.1.3 we discuss how SADL can represent some of the more difficult actions in the Softbot domain. 50 action schemas from the Softbot domain, including all those used in these examples, can be found in Appendix B.

7.1.1 PUCCINI *Goals*

As a demonstration of expressiveness of SADL and the effective of PUCCINI, we presented the following goals to the Softbot, all of which it was able to solve quickly, with very little domain-dependent search control.[1]

1. Find a file named `paper.tex` and containing the string "LCW," and rename the file to `kr.tex` (cf. Section 3.2.2).

2. Print the file *paper*, but recompress it afterward (iff it was compressed initially) (cf. Section 3.2.2).

[1] Five search control rules, each two lines long

3. Find the phone number of the person named Oren Etzioni. (*This goal is solvable by using the University of Washington staff directory,* `staffdir`.)

4. Find the phone number of Leslie Kaelbling, a CS professor at Brown University. (*Solving this goal requires using* `netfind`.)

5. Display all web pages referenced by hyperlinks from both Dan Weld's homepage *and* Oren Etzioni's homepage. (*This goal requires consulting Ahoy! to find the two homepages, reading in both pages to find links in common, and then running Netscape on each common link.*)

6. Compress all files larger than the file named `bigfile`."

7. Make all files in the current directory group-writable.

8. Display the file `doc.tex`. (*This requires running* `latex` *to produce the file* `doc.dvi` *and then running* `xdvi` *to display it*)

9. Print the file `color-picture` to a research printer, making sure that the printout will be color, and report the status of the print job. (*The printout will be color if a color image is printed to a color printer, so the planner must make sure the printer it selects is color. Executing* `lpq` *after the file has been sent to the print queue will reveal the job's status.*)

10. Fetch all files stored at a remote FTP site, and copy them to a local directory.

Table 7.1 shows the planner statistics for solving these goals, including both CPU time and real time for planning and for executing actions. It is traditional just to show CPU time as a measure of the time required to perform a computation, since other processes running concurrently with the one being measured make real time a less-than-reliable metric. The reason we show both measures, instead of just CPU

time, is that CPU time is a poor measure of the time required to execute actions, as Table 7.1 shows quite clearly. From the Softbot's perspective (and ours), executing a UNIX command or fetching a Web page consists mostly of waiting for a reply, which is not computationally expensive. However, the user still must wait that amount of time, which is all that really matters.

We can see from Table 7.1 that all these goals finish fairly quickly and don't involve much search. Only two of the goals, 4 and 5, take more than a minute to solve, and these are both dominated by execution time. There is room for improvement in the efficiency of the PUCCINI planner; the current (Lisp) implementation requires 25 ms of CPU time per plan. However, looking at the real time required for execution reveals that the gains obtainable from speeding up the planner are limited. More substantial gains could be obtained by allowing multiple actions to be executed simultaneously and by continuing to plan while actions are executing.

7.1.2 Plan trace

Below we show a plan trace of the Softbot solving goal 4 from the Table 7.1: finding the phone number of Professor Leslie Pack Kaelbling, from Brown University in Rhode Island. The Softbot solves this goal by using Netfind to find a possible e-mail address for Leslie Kaelbling at Brown, and using finger to verify the e-mail address and to get her `.plan` file, from which it can extract her phone number. The goal given to the Softbot was:

```
(and (satisfy (firstname ?lpk "Leslie"))
                (satisfy (lastname ?lpk "Kaelbling"))
                (satisfy (organization ?lpk "Brown University"))
                (satisfy (state ?lpk "RI"))
                (satisfy (field ?lpk "CS"))
```

Table 7.1: Planner statistics for ten sample goals

prob num.	plans considered	actions executed	planning CPU (s)	execution CPU (s)	execution real time (s)	planning real time (s)
1	53	3	1.306	0.355	1.458	1.623
2	40	3	0.529	0.056	0.376	0.763
3	19	1	0.442	0.105	0.663	0.482
4	721	6	19.721	0.672	38.904	22.945
5	198	8	7.351	3.083	86.44	8.057
6	190	12	5.384	1.017	6.660	7.583
7	565	10	18.426	1.073	3.884	20.320
8	268	6	4.565	0.190	5.294	5.687
9	137	3	2.034	0.139	1.3065	2.092
10	797	6	14.038	1.006	13.28	15.957

```
(satisfy (office.phone ?lpk ?phone))))
```

For clarity, the following plan trace omits plans PUCCINI considered that did not lead to the goal. When multiple choices exist, the one chosen is indicated by a "$\Longrightarrow$"

1. plan = (:GOAL)
2. Add step NETFIND for open goal (SATISFY (ORGANIZATION ?LPK "Brown University")) of goal step
3. plan = (NETFIND :GOAL)
4. Link to step 0 for open goal (SATISFY (CURRENT.SHELL CSH)) of step 1 (netfind)
5. Link to step 0 for open goal (SATISFY (MACHINE.NETFIND.SERVER ?SERVER)) of step 1 (netfind)
6. Reduce OR goal: (OR (FIELD ?PERSON ?KEY3) (EQ ?KEY3 "")) of step 1 (netfind)
   (a) $\Longrightarrow$ Choose disjunct (FIELD ?PERSON ?KEY3))
   (b) Choose disjunct (EQ ?KEY3 "")
7. Reduce OR goal: (OR (CONTEMPLATE (SERVER.BUSY ?SERVER ?TIME) U ((?SERVER))) (AND (CONTEMPLATE (SERVER.BUSY ?SERVER ?TIME)) (CONTEMPLATE (SECONDS-AGO ?TIME 300)))) of step 1 (netfind)
   (a) Choose disjunct $\Longrightarrow$ (CONTEMPLATE (SERVER.BUSY ?SERVER ?TIME) U ((?SERVER)))
   (b) Choose disjunct of (AND (CONTEMPLATE (SERVER.BUSY ?SERVER ?TIME)) (CONTEMPLATE (SECONDS-AGO ?TIME 300)))
8. Link to step 0 for open goal (CONTEMPLATE (SERVER.BUSY ?SERVER ?TIME) U ((?SERVER))) of step 1 (netfind)
9. Link to step 1 for open goal (FIELD ?PERSON ?KEY3) of step 1 (netfind)
10. Add step NETFIND for open goal (FIELD ?PERSON ?KEY3) of step 1 (netfind)
11. Reduce OR goal: (OR (CITY ?PERSON ?KEY1) (ORGANIZATION ?PERSON ?KEY1) (AFFILIATION ?PERSON ?KEY1)) of step 1 (netfind)
    (a) Choose disjunct (CITY ?PERSON ?KEY1)
    (b) $\Longrightarrow$ Choose disjunct (ORGANIZATION ?PERSON ?KEY1)
    (c) Choose disjunct (AFFILIATION ?PERSON ?KEY1)

12. CHOOSE:
    (a) $\Longrightarrow$ Link to step 1 for open goal (AFFILIATION ?PERSON ?KEY1) of step 1 (netfind)
    (b) Add step NETFIND for open goal (AFFILIATION ?PERSON ?KEY1) of step 1 (netfind)
13. Reduce OR goal: (OR (COUNTRY ?PERSON ?KEY2) (STATE ?PERSON ?KEY2) (EQ ?KEY2 "")) of step 1 (netfind)
    (a) Choose disjunct (COUNTRY ?PERSON ?KEY2)
    (b) $\Longrightarrow$ Choose disjunct (STATE ?PERSON ?KEY2)
    (c) Choose disjunct (EQ ?KEY2 "")
14. CHOOSE
    (a) $\Longrightarrow$ Link to step 1 for open goal (ORGANIZATION ?PERSON ?KEY1) of step 1 (netfind)
    (b) Add step NETFIND for open goal (ORGANIZATION ?PERSON ?KEY1) of step 1 (netfind)
15. CHOOSE
    (a) $\Longrightarrow$ Link to step 1 for open goal (SATISFY (FIELD ?LPK "Cs")) of goal step
    (b) Add step NETFIND for open goal (SATISFY (FIELD ?LPK "Cs")) of goal step
16. CHOOSE
    (a) $\Longrightarrow$ Link to step 1 for open goal (STATE ?PERSON ?KEY2) of step 1 (netfind)
    (b) Add step NETFIND for open goal (STATE ?PERSON ?KEY2) of step 1 (netfind)
17. CHOOSE
    (a) $\Longrightarrow$ Add step FINGER for open goal (PERSON.DOMAIN ?PERSON !DOMAIN) of step 1 (netfind)
    (b) Add step STAFFDIR for open goal (PERSON.DOMAIN ?PERSON !DOMAIN) of step 1 (netfind)
18. plan = (FINGER NETFIND :GOAL)
19. Link to step 0 for open goal (SATISFY (CURRENT.SHELL CSH)) of step 2 (finger)
20. Reduce OR goal: (OR (LASTNAME ?PERSON ?STRING) (FIRSTNAME ?PERSON ?STRING) (AND (USERID ?PERSON ?STRING ?DOMAIN) (EQ ?STRING E: !USERID::(?PERSON)))) of step 2 (finger)
    (a) Choose disjunct (LASTNAME ?PERSON ?STRING)
    (b) Choose disjunct of (FIRSTNAME ?PERSON ?STRING)

   (c) Choose disjunct of (AND (USERID ?PERSON ?STRING ?DOMAIN) (EQ ?STRING E: !USERID))

21. CHOOSE
    (a) $\Longrightarrow$ Link to step 2 for open goal (USERID ?PERSON ?STRING ?DOMAIN) of step 2 (finger)
    (b) Add step FINGER for open goal (USERID ?PERSON ?STRING ?DOMAIN) of step 2 (finger)
    (c) Add step STAFFDIR for open goal (USERID ?PERSON ?STRING ?DOMAIN) of step 2 (finger)
22. CHOOSE
    (a) $\Longrightarrow$ Link to step 2 for open goal (PERSON.DOMAIN ?PERSON ?DOMAIN) of step 2 (finger)
    (b) Add step FINGER for open goal (PERSON.DOMAIN ?PERSON ?DOMAIN) of step 2 (finger)
    (c) Add step STAFFDIR for open goal (PERSON.DOMAIN ?PERSON ?DOMAIN) of step 2 (finger)
23. CHOOSE
    (a) $\Longrightarrow$ Link to step 2 for open goal (USERID ?PERSON !USERID !DOMAIN) of step 1 (netfind)
    (b) Add step FINGER for open goal (USERID ?PERSON !USERID !DOMAIN) of step 1 (netfind)
    (c) Add step STAFFDIR for open goal (USERID ?PERSON !USERID !DOMAIN) of step 1 (netfind)
24. CHOOSE
    (a) $\Longrightarrow$ Link to step 1 for open goal (SATISFY (STATE ?LPK "Rhode Island")) of goal step
    (b) Add step NETFIND for open goal (SATISFY (STATE ?LPK "Rhode Island")) of goal step
25. CHOOSE
    (a) Link to step 0 for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)
    (b) Link to step 0 for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)

(c) $\Longrightarrow$ Add step PING for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)
(d) Add step HOSTNAME for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)
(e) Add step HINFO for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)
(f) Add step USERID-LOGIN-MACHINES for open goal (SATISFY (MACHINE.NAME ?MACHINE ?NAME)) of step 2 (finger)

26. plan = (PING FINGER NETFIND :GOAL)
27. Reduce OR goal: (OR (AND (MACHINE.NAME !MACHINE ?MACHINE-NAME) (DOMAIN.MACHINE.NAME !DOMAIN !NAME)) (AND (MACHINE.DOMAIN !MACHINE !DOMAIN) (CURRENT.DOMAIN !DOMAIN) (DOMAIN.NAME.NICKNAME !DOMAIN !NAME !NICKNAME) (MACHINE.NICKNAME !MACHINE ?MACHINE-NAME))) of step 3 (ping)
    (a) $\Longrightarrow$ Choose disjunct (AND (MACHINE.NAME !MACHINE ?MACHINE-NAME) (DOMAIN.MACHINE.NAME !DOMAIN !NAME))
    (b) Choose disjunct (AND (MACHINE.DOMAIN !MACHINE !DOMAIN) (CURRENT.DOMAIN !DOMAIN) (DOMAIN.NAME.NICKNAME !DOMAIN !NAME !NICKNAME) (MACHINE.NICKNAME !MACHINE ?MACHINE-NAME))
28. CHOOSE
    (a) Link to step 0 for open goal (MACHINE.ALIVE !MACHINE) of step 3 (ping)
    (b) $\Longrightarrow$ Link to step 3 for open goal (MACHINE.ALIVE !MACHINE) of step 3 (ping)
    (c) Add step PING for open goal (MACHINE.ALIVE !MACHINE) of step 3 (ping)
29. CHOOSE
    (a) Link to step 0 for open goal (SATISFY (MACHINE.NAME ?SERVER ?SERVER-NAME)) of step 1 (netfind)
    (b) $\Longrightarrow$ Link to step 3 for open goal (SATISFY (MACHINE.NAME ?SERVER ?SERVER-NAME)) of step 1 (netfind)
30. CHOOSE
    (a) Link to step 1 for open goal (SATISFY (FIRSTNAME ?LPK "Leslie")) of goal step
    (b) $\Longrightarrow$ Link to step 2 for open goal (SATISFY (FIRSTNAME ?LPK "Leslie")) of goal step

(c) Add step NETFIND for open goal (SATISFY (FIRSTNAME ?LPK "Leslie")) of goal step
(d) Add step FINGER for open goal (SATISFY (FIRSTNAME ?LPK "Leslie")) of goal step
(e) Add step STAFFDIR for open goal (SATISFY (FIRSTNAME ?LPK "Leslie")) of goal step

31. CHOOSE
    (a) Link to step 1 for open goal (SATISFY (LASTNAME ?LPK "Kaelbling")) of goal step
    (b) $\Longrightarrow$ Link to step 2 for open goal (SATISFY (LASTNAME ?LPK "Kaelbling")) of goal step
    (c) Add step NETFIND for open goal (SATISFY (LASTNAME ?LPK "Kaelbling")) of goal step
    (d) Add step FINGER for open goal (SATISFY (LASTNAME ?LPK "Kaelbling")) of goal step
    (e) Add step STAFFDIR for open goal (SATISFY (LASTNAME ?LPK "Kaelbling")) of goal step

32. CHOOSE
    (a) $\Longrightarrow$ Link to step 2 for open goal (LASTNAME ?PERSON ?LAST) of step 1 (netfind)
    (b) Link to step 1 for open goal (LASTNAME ?PERSON ?LAST) of step 1 (netfind)
    (c) Add step NETFIND for open goal (LASTNAME ?PERSON ?LAST) of step 1 (netfind)
    (d) Add step FINGER for open goal (LASTNAME ?PERSON ?LAST) of step 1 (netfind)
    (e) Add step STAFFDIR for open goal (LASTNAME ?PERSON ?LAST) of step 1 (netfind)

33. CHOOSE
    (a) Add step STAFFDIR for open goal (SATISFY (OFFICE.PHONE ?LPK ?PHONE)) of goal step
    (b) $\Longrightarrow$ Add step INFER-OFFICE-PHONE-FROM-FINGER-REC for open goal (SATISFY (OFFICE.PHONE ?LPK ?PHONE)) of goal step
    (c) Add step INFER-OFFICE-PHONE-SHARED for open goal (SATISFY (OFFICE.PHONE ?LPK ?PHONE)) of goal step

34. CHOOSE
    (a) Link to step 0 for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)
    (b) Link to step 0 for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)
    (c) $\Longrightarrow$ Link to step 3 for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)

(d) Add step PING for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)

(e) Add step HOSTNAME for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)

(f) Add step HINFO for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)

(g) Add step USERID-LOGIN-MACHINES for open goal (MACHINE.NAME !MACHINE ?MACHINE-NAME) of step 3 (ping)

35. plan = (INFER-OFFICE-PHONE-FROM-FINGER-REC PING FINGER NETFIND :GOAL)
36. Execute step NETFIND
37. Link to step 0 for open goal (SATISFY (DOMAIN.MACHINE.NAME ?DOMAIN ?NAME)) of step 2 (finger)
38. Link to step 0 for open goal (DOMAIN.MACHINE.NAME !DOMAIN !NAME) of step 3 (ping)
39. Execute step PING → "cs.brown.edu is alive"
40. Link to step 0 for open goal (SATISFY (MACHINE.ALIVE ?MACHINE)) of step 2 (finger)
41. Execute step FINGER →

```
"Login name: lpk                         In real life: Leslie Pack Kaelbling
Office:  521 Watson,  863-7637          Home phone:  508-520-7826
Directory: /u/lpk                       Shell: /cs/bin/tcsh
Logged in on  Since     Idle   Owner                  Location
     neplus   Jun 25   63:21   Leslie Kaelbling       521
Last login Fri Jun 27 11:53
New mail received Mon Jul 21 05:11:44 1997
     Unread since Fri Jul 18 15:39:44 1997
Plan:
Back from sabbatical!

Brown Office:
        521 CIT
        Computer Science Department
        Box 1910
        Brown University
```

```
          Providence, RI 02912-1910
          401-863-7637 (phone)
          401-863-7657 (fax)
          Email: lpk@cs.brown.edu

Home:     12 Mountain Rock Lane
          Norfolk, MA 02056
          508-520-7826
"
```

42. Expand universals to disjunction after execution
43. CHOOSE
    (a) ⟹ Link to step 0 for open goal (SATISFY (FINGER.RECORD ?PERSON ?F-REC ?DOMAIN)((?PERSON))) of step 4 (infer-office-phone-from-finger-rec))
    (b) Add step FINGER for open goal (SATISFY (FINGER.RECORD ?PERSON ?F-REC ?DOMAIN) ((?PERSON))) of step 4 (infer-office-phone-from-finger-rec))
44. Execute step INFER-OFFICE-PHONE-FROM-FINGER-REC → "863-7637"
45. Planner success!

7.1.3 PUCCINI *Actions*

SADL, like UWL, came out of an effort to build an agent that can "understand" UNIX. Many of the UNIX commands were hard to represent, and the encodings of them went through many generations. We discuss two of the trickier actions, among those used in the previous example.

`Finger`

`finger` takes an argument that may be a userid or the user's first or last name. It returns information about all users whose userid, first name or last name happens to be the string in question. The information returned includes the full name and userid of the user along with other information, which often includes the user's phone number. This wealth of information and flexibility of use makes `finger` quite useful,

but also challenging to model. The reason is that `finger` can be thought of as providing a user's name given her userid, a user's userid given her first or last name, her first name given her last name, and so on. An additional complication is that `finger` takes an optional domain argument, separated from the first argument by a "@." if the domain name is missing, `finger` returns information about users in the "current" domain, that is, the domain of the machine on which `finger` is executed. Despite the best efforts of a number of intelligent people who thought long and hard on this matter, it proved impossible to represent `finger` in the precursor to SADL using fewer than *six* operators, representing different ways `finger` can be used. Even these six didn't capture all the possibilities. The problem stemmed from confusion about knowledge preconditions. Once one starts thinking in terms of knowledge preconditions, there are all sorts of paradoxical problems that emerge. For example, most of the "sensor" commands we've looked at can be thought of as either requiring or providing some piece of information, but not both. If I have some person in mind and I want to find out whether that person is logged on, I could use `finger`. But then `finger` obviously has the precondition of knowing that person's userid. If I have some userid in mind and I want to find out what person has that userid, I could use `finger` for that purpose, but now `finger` had better not have the precondition of knowing that person's userid, since then I'll have a goal-stack cycle.

The solution, as mentioned in Section 3.2.6, is to do away with knowledge preconditions, and write the action instead using conditional effects. Doing so allowed us to replace the six `finger` operators with one. We obtained similar results for other operators as well. See Appendix B.2 for our version of `finger`.

`Netfind`

`Netfind` is a program for finding a possible e-mail address of a person given information about that person, including last name and either institution or sufficiently precise geographic location. `Netfind` has a terrible user interface and is painfully slow,

which makes it a good tool to take away from users and give to agents. `Netfind` has been largely superseded by the Web, but the corresponding Web resources could be represented in essentially the same way.

The most interesting aspect of `Netfind` is that it only returns a guess of the correct e-mail address, so the information must be verified by some other way, such as the UNIX finger command. In fact, after executing `Netfind`, we don't even have confirmation that the information provided is about the person we're looking for – all we know is that some anonymous individual with the given last name and given location possibly has the email address returned. This creates a serious representational challenge, since `Netfind` provides no useful information until its guess has been verified, but needs to be executed to provide the guess. In the precursor to SADL, this paradox was handled by splitting `Netfind` into two actions, the first of which was executed to generate the guess given to finger, and the second of which was executed after finger to assert the location information for that person. The second action didn't actually *do* anything – it just provided a way to temporally separate the effects of `Netfind`.

Using SADL, we can represent `Netfind` more simply by using conditional effects with unannotated preconditions, asserting that the location information about that person holds, provided that the person has the given e-mail address. PUCCINI can use assumptions to support these preconditions using the effects of finger, even though finger is executed after `Netfind`. See Appendix B.2 for the encoding of `Netfind`.

7.2 LCW

In the previous sections, we argued that our LCW mechanism is computationally tractable, but incomplete. However, asymptotic analysis is not always a good predictor of real performance, and incompleteness is a matter of degree. To evaluate our LCW machinery empirically, we measured its impact on the performance of the

Internet Softbot [26].[2]

In this section, we address the following questions experimentally:

- **Speed:** What is the speed of LCW queries and LCW updates as a function of the size of the LCW database and the size of LCW formulas?

As shown in Section 7.2.1, LCW inference is very fast, 2 milliseconds per query, and updates are even faster: 1.2 milliseconds. Times increase for longer queries, but are relatively unaffected by the size of $\mathcal{L}$ and $\mathcal{M}$.

- **Completeness:** Because our LCW database is incomplete, a query may result in the truth value `U` even though its "true" truth value is `F` (Figure 2.1). How often does this occur as the database processes a sequence of queries and updates issued by the planner?

Section 7.2.2 argues that the incompleteness of our LCW mechanism is more of a theoretical concern than a practical one. In over 99% of the cases that occur in practice, the LCW mechanism deduces the correct answer.

- **Impact:** What is the effect of the LCW machinery on the speed with which the planner can control the Internet Softbot? In particular, does the use of LCW information improve the agent's performance enough to offset the cost of LCW inference and update?

Even though LCW inference is fast and effectively complete, it is still conceivable that its use might detract from an agent's overall performance. Section 7.2.3 shows that this is not the case; indeed, LCW's ability to focus search and eliminate redundant sensing operations yields a 100-fold improvement in overall performance.

---

[2] The following data were collected using the XII planner, the immediate predecessor of PUCCINI; the only significant difference between the two planners is support for SADL extensions, which has no bearing on these experiments. The LCW machinery is identical.

7.2.1 Factors Influencing LCW Speed

The interesting questions regarding LCW speed are "How fast are queries and updates on average?" and "How does the time vary as a function of the length of the LCW formula and the size of $\mathcal{L}$ and $\mathcal{M}$?" To answer these questions we randomly generated several thousand goals as explained in Section 7.2.4. In the course of solving these planning problems, the planner issued over 390,000 LCW queries and performed numerous updates. On average, answering an LCW query required 2 milliseconds while processing an update took 1.2 milliseconds.

In answer to the second question, Figure 7.1 shows query time as a function of the length of the query and the size of the $\mathcal{L}$ database.[3] The graph shows the results for query sizes of up to four conjuncts; larger queries don't occur in our softbot's domain. In fact, even queries with four conjuncts occur only as a result of user-supplied $\forall$ goals. The slow growth of query time as a function of $|\mathcal{L}|$ is due to the use of hashing, as opposed to the more expensive linear-time search assumed in our complexity analysis (Section 3.5.7). As mentioned earlier, updates are even faster than queries on average.

7.2.2 Completeness

Because our LCW machinery is incomplete, `QueryLCW`(Φ) may return "No" when the agent does in fact have LCW(Φ). We refer to this event as an LCW *miss*. Below, we explain how we measured the percentage of LCW queries that result in LCW misses.

The problem of detecting LCW misses raises a thorny issue. LCW is a semantic notion defined in terms of $\mathcal{S}$, the infinite set of possible world states that are consistent with the agent's observations. How can we measure, experimentally, the percentage of times when the agent ought to have LCW, but does not? Comprehending the

---

[3] The size of $\mathcal{M}$ is strongly correlated with the size of $\mathcal{L}$, resulting in a very similar graph of query time with respect to the size of $\mathcal{M}$.

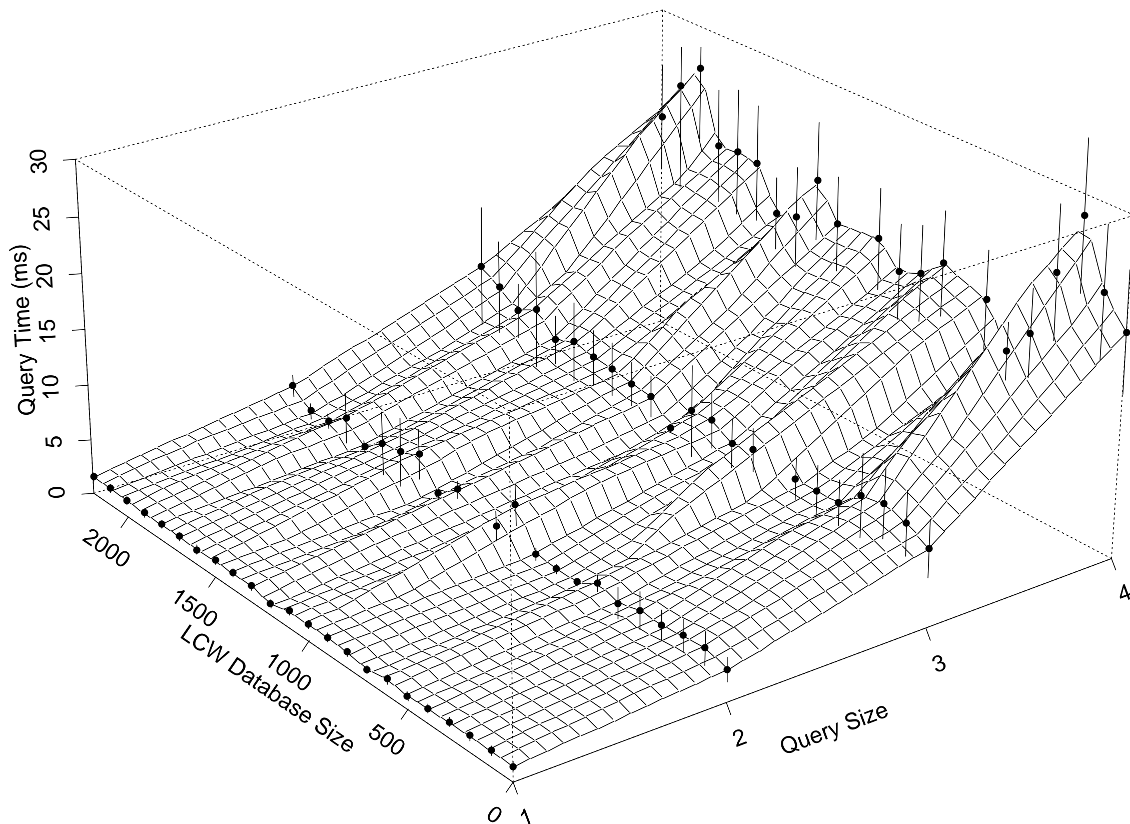

Figure 7.1: CPU Time for LCW queries as a function of the size of the LCW database $\mathcal{L}$, and the number of conjuncts in the query. Experiments were run on a Sun SPARC-station 20; vertical bars indicate 95% confidence intervals. Note that even as $\mathcal{L}$ grows large, the average query time is approximately 2 milliseconds. Over 90% of the 390,000 queries contained fewer than three conjuncts. Because the sizes of $\mathcal{L}$ and $\mathcal{M}$ are strongly correlated in all of our experiments, the graph of query time with respect to the size of $\mathcal{M}$ is similar, and thus omitted.

answer to this question requires a deep understanding of the formal basis for LCW. The definition of LCW in Section 2.2, combined with the fact that if $\varphi \in \mathcal{M}$ then $\mathcal{S} \models \varphi$, implies that if LCW($\Phi$) then there is a one-to-one correspondence between instances of Φ in $\mathcal{W}$ and in $\mathcal{M}$.

This one-to-one correspondence is important because it can be tested experimentally via simulation. Section 7.2.4 describes the methodology in more detail, but the idea is simple. We replace the agent's effectors (which normally manipulate an actual UNIX shell), with new procedures that update and sense a simulation of a UNIX computer. Although the simulated environment doesn't model *every* aspect of UNIX, it is complete relative to every action that could be executed in service of the test suite.

Thus, to check whether `QueryLCW(`Φ`)` has resulted in an LCW miss, we do the following: When `QueryLCW` returns "No," we check whether every instance of Φ in the simulation in fact appears in $\mathcal{M}$. If so, LCW is possible, and we report that an LCW miss has occurred. Of course, this mechanism can over-report LCW misses. Although LCW(Φ) is *possible*, and `QueryLCW(`Φ`)` failed, it may be that no sensing of Φ has taken place and we could not expect any agent to deduce LCW(Φ).

For example, if directory `dir1` is empty, then both $\mathcal{M}$ and the simulation database will agree on the extension of `in.dir(dir1,` f`)`, even if the agent has never executed a command such as `ls dir1`. But not knowing whether there are any files in a directory that happens to be empty is not the same as knowing that there aren't any, so this case would be a *false miss.* We are able to eliminate some of these false misses, but not all of them. However, since we are trying to demonstrate the success of our LCW machinery, we are content to be conservative and overstate the number of LCW misses.

In our experiments, fewer than 1% of the LCW queries generated by the planner result in misses. The percentage of misses does not vary significantly with the amount of dynamism, or with the percentage of Domain Growth or Information Loss updates

that occur.

Answering the question of how often misses occur independent of the XII planner and the Softbot domain is problematic, since we could construct cases in which all LCW queries are misses, or none are. For example, suppose we have a directory containing only postscript and TeX files, and we have LCW on the size of all files in that directory. Suppose we then compress one of the postscript files. By the Information Loss Rule, the LCW we had on the size of all the files will be removed from $\mathcal{L}$, whereas if our LCW machinery were *complete*, it would *retain* LCW on the size of all TeX files in the directory. Now if all queries are of the form "Do I know the size of all TeX files in this directory?" then every query will be a miss. Perverse cases like this one in practice are highly unlikely. This is due, in part, to the fact that failed LCW queries are likely to be followed by actions that achieve the desired LCW.

7.2.3 Impact on Planning

We have shown that individual LCW queries are fast and that the reasoning mechanism is effectively complete, but given that a significant number of LCW queries are performed during planning, it is still conceivable that LCW might slow the planner down. We show that this is not the case; in fact LCW inference speeds planning considerably by reducing redundant sensing operations. Figure 7.2 shows the performance of the XII planner with and without LCW, solving a sequence of randomly generated goals, with $\mathcal{M}$ and $\mathcal{L}$ initially empty. The planner runs faster with LCW even on the first goal, since it leverages the LCW information which it gains in the course of planning. In subsequent goals, the planner can take advantage of LCW gained in previous planning sessions for an even more pronounced speedup. Without LCW, the planner wastes an enormous amount of time doing redundant sensing. The version of XII without LCW completed only 8% of the goals before hitting a fixed time bound of 1000 CPU seconds. In contrast, the version with LCW completed 94% of the goals in the allotted time.

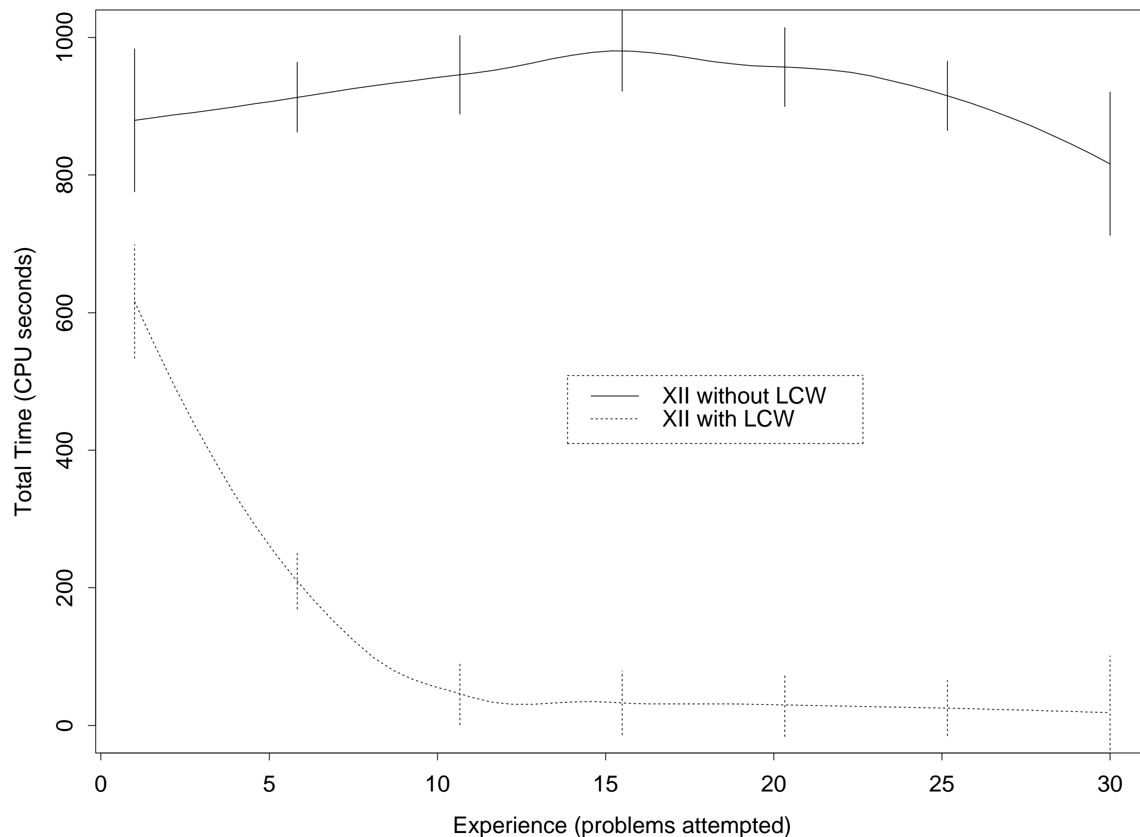

Figure 7.2: The use of LCW reasoning yields dramatic performance improvements to the XII planner. Times indicated are CPU seconds on a Sun SPARCstation 20; vertical bars indicate 95% confidence intervals. The experiment was repeated 10 times on randomly generated initial worlds. Thus, each of 30 distinct points on the X-axis represents the average of 10 planning sessions on randomly generated goals. The databases $\mathcal{M}$ and $\mathcal{L}$ were left intact between goals to measure the impact of increasing knowledge on planner performance. Thus, $\mathcal{M}$ and $\mathcal{L}$ tend to increase along the X-axis. The curves show a best-fit to each set of 300 data points.

Table 7.2: The number of executions performed by the planner with and without LCW on 300 randomly generated problems. The number of executions for the planner without LCW are drastically under-reported, because without LCW the planner could only solve 8% of the problems within the 1000 CPU second time bound. In contrast, with LCW reasoning the planner solved 94% of the problems. Had both versions of the planner been run until every problem was solved, we would expect a much larger difference in favor of the planner with LCW. Surprisingly, LCW also reduces the amount of time XII spends per plan on average. This is because the non-LCW planner tends to consider more complicated plans, which require more CPU time to evaluate.

LCW?	% probs solved	Total Number of executed actions	Time per plan (sec)
yes	94%	34,865	0.26
no	8%	93,050	1.16

7.2.4 The Experimental Framework

The goal of our experiments was to measure the performance of our LCW machinery in a real-world setting. All of our evaluations of LCW are through queries and updates generated by the XII planner in the course of satisfying randomly-generated file manipulation goals in the Softbot domain. To make our experiments easier to control, vary, and replicate, we built a simulation environment that allows us to generate arbitrary UNIX world states, which behave exactly as UNIX behaves in response to actions executed by the softbot. Additionally, the simulation greatly simplifies the task of evaluating LCW, as we discuss in Section 7.2.2. Nearly all of the results we report using simulated UNIX worlds are identical to the results we would obtain if XII were executing in an equivalent, real UNIX environment. The one exception is the report of total time in Figure 7.2, which does not reflect the time required to execute actions in a UNIX shell. However, the purpose of Figure 7.2 is to evaluate the impact of LCW on planning, not to measure the performance of the Internet Softbot. Based on earlier experiments in this domain (see [35]), it seems

likely that accurately reporting execution time would only make our results stronger, since, without LCW, XII spends a greater percentage of its time executing actions (see Table 7.2), and execution is expensive.

7.2.5 The Simulation Environment

The simulation environment consists of a current world state w_s, represented as a database, which completely specifies the state of all files and directories in the simulation, and an execution procedure that translates an action to be executed into the appropriate queries and updates on w_s. In our experiments, w_s contains up to 80 directories, each directory holding between 5 and 20 files. The topology of the directory tree is random, each directory containing at most five other directories. Filenames are all of the form `dir1`, `file2`, etc. The values of other file attributes, such as `size` and `file.type`, are chosen randomly. Although w_s doesn't model *every* aspect of UNIX, it is complete relative to every action that could be executed in service of the test suite.

The execution procedure simply computes a mapping from an action to database operations on w_s. This mapping is straightforward; all the required information is contained in the effects of the action. For example, `ls -la dir3` determines, among other things, the size of each file in `dir3`, so the execution procedure handles the execution of `ls -la dir3` by querying w_s for

$$\texttt{in.dir}(f,\ \texttt{dir3}) \wedge \texttt{size}(f,\ n)$$

and updating $\mathcal{M}$ with the results. Similarly, since `cd dir11` has the effect `current.dir (dir11)`, this update is done to w_s as well as to $\mathcal{M}$.

7.2.6 The Goal Distribution

The test suite consists of a series of *runs*. At the beginning of each run, a simulated world w_s is randomly generated, and $\mathcal{M}$ and $\mathcal{L}$ are empty. A sequence of 30 goals is

then randomly generated, and the planner is given the goals to solve one by one. $\mathcal{M}$ and $\mathcal{L}$ are left intact between goals, so for each goal, the planner has the benefit of knowledge obtained in solving the previous goals. After the 30 goals are completed, a new world is generated, $\mathcal{M}$ and $\mathcal{L}$ are emptied, and the process is repeated.

Our goal generator creates either universally-quantified or existentially-quantified goals. Quantification aside, the two sets of goals are essentially equivalent, and consist of finding files meeting certain properties, such as `filename`, `in.dir`, `word.count` and `file.type`, and performing certain operations on them, such as compressing them, moving them to a different directory or finding out their size. A typical goal is "Compress all postscript files in the directory `/dir0/dir1/dir21`."

Chapter 8

CONCLUSIONS

The work presented in this thesis seeks the middle ground between planning and knowledge representation, between expressiveness and tractability, and between theoretical and empirical results. This quest for the middle ground is borne from practical necessity. When building complete agents, many of the representational assumptions made in planning must be reconsidered. For agents to be useful, they must support expressive representations for reasoning about the world, but they must also perform this reasoning quickly. Finally, to have confidence in our agents, we want both formal guarantees that the behavior of our agents is correct, and empirical verification that they are effective. We have presented three interrelated contributions to planning and knowledge representation:

- The LCW representation of local closed world knowledge, along with sound algorithms for performing inference and updates on this knowledge

- The SADL action language for representing information goals and sensors that return an unbounded amount of information

- The PUCCINI planning algorithm, which uses LCW and SADL to solve a wide class of goals, including information goals and universally quantified goals, in the presence of incomplete information.

8.1 Related Work

8.1.1 PUCCINI

PUCCINI is an extension of XII, which is based on the UCPOP algorithm [72]. PUCCINI builds on XII by supporting SADL and handling assumptions. XII builds on UCPOP by supporting UWL, dealing with information goals and effects, interleaving planning with execution and LCW. The algorithm we used for interleaving planning with execution closely follows IPEM [1]. PUCCINI differs from IPEM in that PUCCINI can represent actions that combine sensing and action, and can represent information goals as distinct from satisfaction goals. IPEM makes no such distinction, and thus cannot plan for information goals. OCCAM [49], and SAGE [43] can both create plans to obtain new information, but unlike PUCCINI, they can do nothing else; both planners are specialized to the problem of information gathering or database retrieval. OCCAM derives significant computational speedup and representational compression from the assumption that actions don't change the world, and thus seems appropriate for domains in which that assumption is valid. It is not clear whether SAGE, which is based on UCPOP as is PUCCINI, gains similar advantage. PUCCINI, in contrast, can integrate causational and observational actions in a clean way, and in addition, supports Local Closed World reasoning, which SAGE does not support, but a recent version of OCCAM [30] now does.

The SADL language is the synthesis and extension of UWL [27] and the subset of ADL [69] supported by UCPOP. Our research has its roots in the SOCRATES planner [50]. Like PUCCINI, SOCRATES utilized the Softbot domain as its testbed, supported the UWL representation language and interleaved planning with execution. In addition, SOCRATES supported a restricted representation of LCW, which it used to avoid many cases of redundant information gathering. Our advances over SOCRATES include the ability to satisfy universally quantified goals, a more expressive LCW representation, and the machinery for automatically generating LCW effects and for

detecting threats to LCW links.

More broadly, our work is inspired by previous work which introduced and formalized the notion of knowledge preconditions for plans and informative actions [60, 63, 16, 64, 65]. While we adopted an approach that interleaves planning and execution [29, 67, 47], other researchers have investigated contingent or probabilistic planning. Contingent planners [85, 81, 38, 73, 15, 75] circumvent the need to interleave planning with execution by enumerating all possible courses of action and deferring execution until every possible contingency has been planned for. While this strategy is appropriate for safety-critical domains with irreversible actions, the exponential increase in planning time is daunting.

Some planners encode uncertainty in terms of conditional probabilities [48, 15], or Markov Decision Processes [44, 10]. Our approach sacrifices the elegance of a probabilistic framework for an implemented system capable of tackling practical problems.

Robotics researchers have also addressed the problem of planning with actions whose effects are uncertain, exploring combinations of sensory actions and compliant motion to gain information [57, 22, 5, 13]. Recent complaints about "Sensor abuse" [58, 61] suggest that the robotics community is aware of the high cost of sensing and is interested in techniques for eliminating redundant sensing.

PUCCINI's search space can be understood in terms of Kambhampati's Universal Classical Planner (UCP) [40], which unifies plan-space planning with forward-chaining and backward-chaining planning. When PUCCINI works on open conditions or threats, it is following a version of the Refine-plan-plan-space algorithm with bookkeeping constraints and conflict resolution. When PUCCINI executes an action, it is following the Refine-plan-forward-state-space algorithm, though PUCCINI actually executes the action and obtains sensory information, whereas Kambhampati's algorithm only modifies the plan. Another difference is that, to preserve completeness, Kambhampati's algorithm considers adding and "executing" all applicable new actions, whereas PUCCINI will only execute actions that were previously added to the plan.

An approach to planning similar in spirit to PUCCINI's use of LCW is to count the number of relevant ground propositions in the model, before inserting information-gathering actions into the plan, to check whether the desired information is already known [83, 67]. However, this heuristic is only effective when the number of sought-after facts is known in advance. LCW reasoning is more general in that it can also deal with cases when the size of a set (e.g. the number of files in a directory) is unknown.

8.1.2 SADL

McCarthy and Hayes [60] first argued that an agent needs to reason about its ability to perform an action. Moore [63] devised a theory of knowledge and action, based on a variant of the situation calculus with possible-worlds semantics. He provided an analysis of knowledge preconditions, which we discussed earlier, and information-providing effects. Morgenstern [64] generalized Moore's results to express partial knowledge that agents have about the knowledge of other agents (e.g. "John knows what Bill said"), using a substantially more expressive logic, which is syntactic rather than modal. Davis [9] extended Moore's theory to handle contingent plans, though, like Moore, he doesn't discuss actions with indeterminate effects. Levesque [51] offers an exceptionally clean theory of when a plan, with conditionals and loops, achieves a satisfaction goal in the presence of incomplete information. However, Levesque doesn't discuss knowledge goals, and his sensory actions can return only `T` or `F`, and can't change the state of the world. Shoham [82] presents a language, with explicit time, for representing beliefs and communication among multiple agents. Agents can request other agents to perform actions, which can include (nested) communicative actions, but not arbitrary goals. A discrete temporal logic, without $\forall$, is used to represent beliefs.

A number of contingent planning systems have introduced novel representations of uncertainty and sensing actions. C-BURIDAN [48, 14] uses a probabilistic action language that can represent conditional, observational effects, including noisy sen-

sors, and effects that cause information loss. Unlike SADL, the C-BURIDAN language is propositional, and makes no distinction between knowledge goals and goals of satisfaction. Cassandra [75] represents uncertainty as a finite disjunction of possibilities, one of which will be true. Uncertain effects are represented as conditional effects with preconditions labeled :unknown. Cassandra produces a single, loop-free contingent plan guaranteed to solve the goal. It cannot represent actions, like `ls` that return information about an unbounded number of objects, and it cannot produce plans that contain actions which might not succeed, even if the entire plan will achieve the goal. Examples of such plans are searching for a file, trying all combinations to a safe, or dunking two packages, one of which contains a bomb.

8.1.3 LCW

Below, we briefly review the large body of related work on circumscription, autoepistemic logic, and database theory. At the end of this section, we summarize the key differences between this body of work and ours.

The bulk of previous work has investigated the *logic* of closed world reasoning (*e.g.*, [45, 23, 78, 63, 52]), and the semantics of theory updates (*e.g.*, [33, 41, 12]). Results include logical axiomatizations of the closed world assumption (CWA), exploring the relationship between the CWA and circumscription, distinguishing between knowledge base revision and knowledge base update, and more. Although decidable computational procedures have been proposed in some cases (*e.g.*, [32], and the Minimality Maintenance System [77]), they remain intractable. Update procedures have been described that involve enumerating the possible logical models corresponding to a database (e.g., [90, 11]), or computing the disjunction of all possible results of an update [42]. In contrast, we adopt the WIDTIO (When In Doubt Throw It Out [91]) policy. As [19] points out, this method is easy to implement efficiently but has the potential disadvantage that, in the worst case, all knowledge in the database has to be retracted. In fact, we have developed novel rules that enable us to retain closed

world information in the face of most updates. We believe our rules satisfy the update postulates specified in [41] and generalized in [12], but have not attempted a proof. Instead, we prove that our update scheme has polynomial worst case running time (Section 3.5) and we demonstrate experimentally that it is effective in practice (Section 7.2).

Motro [66] uses meta-relations much like LCW formulas to encode local validity and completeness in a database. However, his scheme doesn't support updates, and he gives no complexity results. Levy [53] has pointed out a close relationship between closed-world reasoning and the problem of detecting the independence of queries from updates. However, the computational model in the database literature (Datalog programs) is different from our own. Furthermore, polynomial-time algorithms for this problem are rare in the database literature (*e.g.*, [55] merely reports on decidability). Notable exceptions include Elkan's [20] polynomial time algorithm for conjunctive query disjointness (which is a sufficient condition for query independence), and Elkan's [21] approach for handling *monotonic* updates.

Some excellent analyses of the computational complexity of closed-world reasoning have emerged [7, 19], which show that the different approaches described in the literature are highly intractable in the general case. Stringent assumptions are required to make closed-world reasoning tractable. For example, Eiter and Gottlob [19, page 264] show that propositional Horn theories with updates and queries of bounded size yield polynomial-time algorithms. However, all positive computational tractability results reported in [7, 19] are restricted to *propositional* theories. Motivated by the need for closed-world reasoning in modern planning algorithms, we have formulated a rather different special case where the knowledge bases record first-order information, queries are first-order conjunctions, and updates are atomic.[1]

---

[1] Since we consider formulas with an essentially unbounded number of instances, it is impractical to translate our first-order theories into propositional correlates. Furthermore, as shown in Sections 2.4 and 3.5, local closed-world reasoning makes essential use of first-order constructs such as unification.

Some recent work [54, 30, 18] builds on our LCW work, providing a more expressive representation, though the price of higher complexity bounds. The extensions allow one to state, for example, that formula Φ is complete for all instances satisfying Ψ. All of these systems are designed for pure information gathering, such as database query planning, and thus don't need to address the problem of updating LCW in response to changes in the world.

8.2 Future Work

8.2.1 Contingency

As we discussed, interleaving planning with execution is fundamentally incomplete in the presence of irreversible actions, yet contingency planning is impractical for realistic domains. We hope to overcome these limitations by combining both techniques. The agent would interleave planning with execution when possible, but would plan for contingencies when in danger of falling off a cliff. How best to decide when to plan for contingencies and when to execute is an interesting open problem.

8.2.2 Exogenous Events

Although we have relaxed the assumption of complete information, we still assume correct information. Since we want our agents to cope with exogenous events, we are in the process of relaxing this assumption as well. We are investigating two complementary mechanisms to solve this problem. The first mechanism associates *expiration times* with beliefs. If an agent has a belief regarding φ, which describes a highly dynamic situation (*e.g.*, the idle time of a user on a given machine), then the agent should not keep that belief in $\mathcal{M}$ for very long. Thus, the agent would expect a block that it put on the table at 10:00 to be on the table at 10:05, but it would not have the same expectations for a cat. Thus, after an appropriate amount of time has elapsed, the update $\Delta(\varphi, \mathtt{T} \vee \mathtt{F} \rightarrow \mathtt{U})$ occurs automatically. Note that

by the Information Loss Rule, this update will cause LCW to be retracted as well. Although this representation has the advantage of being simple, there is quite a lot it can't represent, such as the fact that the cat is more likely to remain on the table if there's a saucer of milk present, or the fact that the milk is less likely to remain in the saucer if the cat is present. Furthermore, this mechanism is only effective when the belief about φ expires before φ changes in the world, which we cannot guarantee in general.

Thus, an additional mechanism is required that enables the agent to detect and recover from out-of-date beliefs. This is a harder problem, because it involves belief revision, rather than mere update. If executing an action fails, and the action's preconditions are known, it follows that one or more of the preconditions of the action were not satisfied — but which ones? A conservative mechanism would retract the ground literals satisfying the action's preconditions from the agent's theory. However, this mechanism could discard a great deal of valuable information. We are investigating less conservative mechanisms.

Both of these approaches are somewhat reactive. It would be much more satisfying to have the agent reason explicitly about changes outside its control. This would be useful not only for planning more effectively, but also as an aid in plan recognition. For example, without some theory of exogenous events, there would be no way of explaining why someone would carry an umbrella even though it's not raining.

However, it is not at all clear how best to represent this knowledge. The most straightforward approach would be to assume that anything can change at any time. However, if the agent can never assume that conditions it has achieved remain achieved, it is forced into the obsessive-compulsive practice of repeatedly sensing to make sure "closed" subgoals haven't been clobbered. Furthermore, without some information about the relative likelihood of various exogenous events, the agent might, for example, avoid going outside for fear of getting hit by a meteor. While we could address this problem using probabilities, unless the true probabilities are known, this

approach is simply *ad hoc.* Furthermore, reasoning with probabilities is expensive.

8.2.3 Expressiveness

We need to investigate increasing the expressive power of $\mathcal{M}$ and $\mathcal{L}$. First, the introduction of negation into $\mathcal{L}$ would enable us to express sentences such as "I know the size of each file in `/kr94` except `myfile`," which would make LCW update less conservative. Second, suppose that an agent was unfamiliar with the contents of the `/kr94` directory, yet executed `chmod g+r *` while in that directory. The reasoning mechanism described in this paper is incapable of inferring that all the files in `/kr94` are group-readable.[2] The LCW sentence

$$\textsc{lcw}(\texttt{in.dir}(f, \texttt{/kr94}) \wedge \texttt{group.protection}(f, \texttt{readable}))$$

is not warranted because it implies that the agent is familiar with all the group-readable files in `/kr94`, which is false by assumption.

We *could* represent the information gained from the execution of `chmod g+r *` in `/kr94` by introducing the following Horn clause into $\mathcal{M}$:

$$\texttt{in.dir}(f, \texttt{/kr94}) \rightarrow \texttt{group.protection}(f, \texttt{readable})$$

The Horn clause says that all the files in the directory `/kr94` are group-readable, even though the agent may be unfamiliar with the files in `/kr94`. Although the mechanisms described in this paper do not allow Horn clauses in $\mathcal{M}$, this example demonstrates that such an extension would provide increased expressiveness. Future work should determine whether this increase in expressive power is worthwhile.

8.2.4 Background Goals

Much of the work in planning focuses on "satisficing" search — the agent is given a goal condition, which it must achieve at some future time. Any plan that achieves

[2] Indeed, this inference is only licensed when the agent is authorized to change the protection on each of the files in `/kr94`; suppose this is the case.

the goal is as good as any other, and once the goal is achieved, the agent can call it a day. By these criteria, if the agent is given the goal "delete the file named `core`," executing `rm -rf /*` is just as good as executing `rm core`. The side effect of the former, deleting all the files on the file system, is irrelevant. One way out of this dilemma is to replace satisficing search with optimizing search. The agent is given a utility function that describes the relative desirability of all possible states of the world, and it produces a plan to maximize its utility. While this approach is appropriate for some domains, it has two significant liabilities. The first is obvious: computational cost. The second is less obvious but more problematic: Where do these utility functions come from? Getting users to specify precise goals is hard enough; eliciting utility functions is a even more challenging.

The approach we take in the Softbot is to require users to provide background goals that the Softbot should always take into account while planning. Such background goals can include *safety goals* such as "Don't delete my thesis" and *tidiness goals*, such as "Re-compress my files when you're done printing them." [87] While stating background goals is less work than providing a utility function to the agent, it is still too much work. It would be nice to have a *meta-tidiness* goal of "Keep everything in its place," and a *meta-safety* goal of "Do no harm." But this just pushes the problem into representing the "place" for everything, and defining "harm." Furthermore, it's unlikely that we would want our agents to adhere to these rules too rigidly. There are times, for example, when cleaning up isn't worth the trouble.

It may be possible to come up with a more general solution to the goal of providing background goals by understanding where background goals come from. One possibility is that background goals result from anticipating future goals or exogenous events. The reason I don't execute `rm -rf /*` is that I may need those files in the future. I re-compress my files because otherwise I might run out of disk space some time later. I hang up the phone when I'm done with it because I might get a call later.

It might be possible to figure out appropriate background goals by doing an analysis of the planning domain, possibly by taking sample problems and generating plans for them, and figuring out what conditions are likely to be critical some time in the future.

8.3 Summary

We began by describing our representation of the agent's incomplete information and local closed world knowledge. We defined the semantics of the agent's knowledge using the situation calculus and the knowledge relation, K. We presented inference procedures for LCW, and showed that these procedures are sound and run in polynomial time.

We then presented the SADL action language and provided the semantics using the situation calculus, first defining the **initially**, **satisfy** and **hands-off** goal annotations and discussing universally quantified goals, and then discussing the effect annotations **cause** and **observe**, universally quantified and conditional effects, and effects that introduce uncertainty. We then discussed temporal projection in SADL, and showed how LCW updates result from SADL actions. We presented procedures for performing these updates, and showed that they are sound and run in polynomial time.

We next discussed plans formed from SADL actions and showed how these plans are represented with structures used by the PUCCINI algorithm. We defined a PUCCINI planning problem, and finally presented the PUCCINI planning algorithm, discussed the novel ways of achieving SADL goals, including universally quantified goals and LCW goals, in the presence of incomplete information. We introduced an assumption mechanism based on causal links and introduced novel ways of resolving threats to causal links, including links that protect universally quantified goals, LCW goals and **hands-off** goals. We discussed execution in PUCCINI, and techniques for backtracking over execution when a plan fails.

We then proved that PUCCINI is sound, and discussed the reasons that it is incomplete. Finally, we presented empirical results that show LCW and PUCCINI to be effective.

BIBLIOGRAPHY

[1] J Ambros-Ingerson and S. Steel. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. on AI*, pages 735–740, 1988.

[2] R. Brachman. "Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, October 1992.

[3] D. Brill. *LOOM Reference Manual.* USC-ISI, 4353 Park Terrace Drive, Westlake Village, CA 91361, version 1.4 edition, August 1991.

[4] R. Brooks and L. Stein. Building brains for bodies. *Autonomous Robots*, 1(1):7–25, 1994.

[5] R. Brost. Automatic grasp planning in the presence of uncertainty. *International Journal of Robotics Research*, 7(1):3–17, February 1988.

[6] T. Bylander. Complexity results for planning. In *Proceedings of IJCAI-91*, pages 274–279, 1991.

[7] M. Cadoli and M. Schaerf. A survey of complexity results for non-monotonic logics. *Journal of Logic Programming*, 17:127–160, November 1993.

[8] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Publishing Corporation, New York, NY, 1978.

[9] E. Davis. Knowledge preconditions for plans. Technical Report 637, NYU Computer Science Department, May 1993.

[10] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76, 1995.

[11] Alvaro del Val. Computing Knowledge Base Updates. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 740–750, 1992.

[12] Alvaro del Val and Yoav Shoham. Deriving Properties of Belief Update from Theories of Action (II). In *Proceedings of IJCAI-93*, pages 732–737, 1993.

[13] B. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence*, 37:223–271, 1988 1988.

[14] D. Draper, S. Hanks, and D. Weld. A probabilistic model of action for least-commitment planning with information gathering. In *Proc. 10th Conf. on Uncertainty in Artifical Intelligence*, 1994.

[15] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, June 1994.

[16] M. Drummond. A representation of action and belief for automatic planning systems.

[17] M. Drummond. Situated control rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning*, May 1989.

[18] O. Duschka. Query optimization using local completeness. In *Proc. 14th Nat. Conf. on AI*, 1997.

[19] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artificial Intelligence*, 57:227–270, October 1992.

[20] Charles Elkan. A decision procedure for conjunctive query disjointness. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 134–139, 1989.

[21] Charles Elkan. Independence of logic database queries and updates. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 154–160, 1990.

[22] M. Erdmann. On Motion Planning with Uncertainty. AI-TR-810, MIT AI LAB, August 1984.

[23] D. Etherington. *Reasoning with Incomplete Information.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[24] D. Etherington, S. Kraus, and D. Perlis. Nonmonotonicity and the scope of reasoning: Preliminary report. In *Proc. 8th Nat. Conf. on AI*, pages 600–607, July 1990.

[25] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1–2):113–148, January 1997.

[26] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[27] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

[28] Richard. P. Feynman. *Surely You're Joking, Mr. Feynman.* Bantam Books, New York, 1985.

[29] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

[30] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proc. 15th Int. Joint Conf. on AI*, 1997.

[31] M. Genesereth and I. Nourbakhsh. Time-saving tips for problem solving with incomplete information. In *Proc. 11th Nat. Conf. on AI*, pages 724–730, July 1993.

[32] M. Ginsberg. A circumscriptive theorem prover. *Artificial Intelligence*, 39(2):209–230, June 1989.

[33] M. Ginsberg and D. Smith. Reasoning about action I: A possible worlds approach. *Artificial Intelligence*, 35(2):165–196, June 1988.

[34] K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 174–185, 1996.

[35] Keith Golden, Oren Etzioni, and Dan Weld. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. on AI*, pages 1048–1054, 1994.

[36] Peter Haddawy and Steve Hanks. Utility Models for Goal-Directed Decision-Theoretic Planners. Technical Report 93–06–04, Univ. of Washington, Dept. of Computer Science and Engineering, September 1993. Submitted to *Artificial Intelligence*. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[37] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen and Compandy, Ltd., London, 1968.

[38] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proc. 7th Nat. Conf. on AI.* Morgan Kaufmann, 1988.

[39] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76:167–238, 1995.

[40] S. Kambhampati and B. Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In *Proc. 2nd European Planning Workshop*, 1995.

[41] H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proc. 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 387–394, 1991.

[42] A. Keller and M. Wilkins. On the use of an extended relational model to handle changing incomplete information. *IEEE Transactions on Software Engineering*, SE-11(7):620–633, 1985.

[43] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. 14th Int. Joint Conf. on AI*, pages 1686–1693, 1995.

[44] S. Koenig. Optimal probabilistic and decision-theoretic planning using markovian decision theory. UCB/CSD 92/685, Berkeley, May 1992.

[45] K. Konolidge. Circumscriptive ignorance. In *Proc. 2nd Nat. Conf. on AI*, pages 202–204, 1982.

[46] R. Kowalski. Logic for data description. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–103. Plenum Publishing Corporation, New York, NY, 1978.

[47] K. Krebsbach, D. Olawsky, and M. Gini. An empirical study of sensing and defaulting in planning. In *Proc. 1st Intl. Conf. on AI Planning Systems*, pages 136–144, June 1992.

[48] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, 76:239–286, 1995.

[49] C. Kwok and D. Weld. Planning to gather information. Technical Report 96-01-04, University of Washington, Department of Computer Science and Engineering, January 1996. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[50] Neal Lesh. A planner for a UNIX softbot. Internal report, 1992.

[51] Hector Levesque. What is planning in the presence of sensing? In *Proc. 13th Nat. Conf. on AI*, 1996.

[52] H.J. Levesque. All I know: A study in autoepistemic logic. *Artificial Intelligence*, 42(2–3), 1990.

[53] A. Levy. Queries, updates, and LCW. Personal Communication, 1994.

[54] A. Levy. Obtaining complete answers from incomplete databases. In *Proc. 22nd VLDB Conf.*, 1996.

[55] A. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of the 19th VLDB Conference*, 1993.

[56] V. Lifschitz. Closed-World Databases and Circumscription. *Artificial Intelligence*, 27:229–235, 1985.

[57] T. Lozano-Perez, M. Mason, and R. Taylor. Automatic synthesis of fine motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, Spring 1984.

[58] M. Mason. Kicking the sensing habit. *AI Magazine*, 14(1):58–59, Spring 1993.

[59] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1,2):27–39, April 1980.

[60] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[61] D. Miller. A twelve-step program to more efficient robotics. *AI Magazine*, 14(1):60–63, Spring 1993.

[62] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40(1–3):63–118, 1989. Available as technical report CMU-CS-89-103.

[63] R. Moore. A Formal Theory of Knowledge and Action. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World*. Ablex, Norwood, NJ, 1985.

[64] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, pages 867–874, 1987.

[65] Leora Morgenstern. *Foundations of a Logic of Knowledge, Action, and Communication*. PhD thesis, New York University, 1988.

[66] Amihai Motro. Integrity = validity + completeness. *TODS*, 14(4):480–502, 1989.

[67] D. Olawsky and M. Gini. Deferred planning and sensor use. In *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*. Morgan Kaufmann, 1990.

[68] C. Papadimitriou. Games against nature. *Journal of Computer and Systems Sciences*, 31:288–301, 1985.

[69] E. Pednault. *Toward a Mathematical Theory of Plan Synthesis*. PhD thesis, Stanford University, December 1986.

[70] E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[71] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[72] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

[73] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. on AI Planning Systems*, pages 189–197, June 1992.

[74] Martha Pollack. The uses of plans. *Artificial Intelligence*, 57(1), 1992.

[75] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research*, 1996.

[76] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty.* Addison-Wesley.

[77] O. Raiman and J. de Kleer. A Minimality Maintenance System. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 532–538, October 1992.

[78] R. Reiter. Circumscription implies predicate completion (sometimes). *Proc. 2nd Nat. Conf. on AI*, pages 418–420, 1982.

[79] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[80] R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *Proc. 11th Nat. Conf. on AI*, pages 689–695, July 1993.

[81] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proc. 10th Int. Joint Conf. on AI*, pages 1039–1046, August 1987.

[82] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[83] D. Smith. Finding all of the solutions to a problem. In *Proc. 3rd Nat. Conf. on AI*, pages 373–377, 1983.

[84] A. Tate. Generating project networks. In *Proc. 5th Int. Joint Conf. on AI*, pages 888–893, 1977.

[85] D. Warren. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, pages 344–354, University of Edinburgh, 1976.

[86] D. Weld. An introduction to least-commitment planning. *AI Magazine*, pages 27–61, Winter 1994. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[87] Dan Weld and Oren Etzioni. The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. on AI*, pages 1042–1047, 1994.

[88] D. E. Wilkins. *Practical Planning.* Morgan Kaufmann, San Mateo, CA, 1988.

[89] M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, June 1994.

[90] M. Winslett. Reasoning about action using a possible models approach. In *Proc. 7th Nat. Conf. on AI*, page 89, August 1988.

[91] M. Winslett. *Updating Logical Databases.* Cambridge University Press, 1990. Cambridge, England.

Appendix A

PROOFS

A.1 Proofs from Chapter 2

In many of the following proofs we rely on the following two facts:

- $\mathcal{L}$ contains only positive sentences.

- The variable substitution θ maps a sentence Φ to a ground sentence $\Phi\theta$. Thus, once the truth value of $\Phi\theta$ is known, we have $\text{LCW}(\Phi\theta)$.

A.1.1 Proof of Theorem 2.1 (NP-hardness of LCW queries, unrestricted $\mathcal{L}$)

We reduce formula satisfiability (SAT) to the problem of answering a singleton LCW query. Let π be an arbitrary propositional boolean formula. Let $\mathcal{L} = \{\text{LCW}(p \vee \pi)\}$, where p is a proposition not appearing in π, and let $\mathcal{M}$ be empty. We will show that the query $\text{LCW}(p)$ *fails* iff there is a truth assignment to propositions in π such that π is true. Thus answering $\text{LCW}(p)$ in the negative can be used to determine whether π is satisfiable.

1. Say there is no assignment such that π is true (i.e. π is provably false). Thus $p \vee \pi$ has the same truth value as p, so it follows from the definition of LCW that $\text{LCW}(p \vee \pi)$ implies $\text{LCW}(p)$. Therefore, the query must succeed.

2. Conversely, if π is satisfiable, then either π is provably true (a tautology), or neither π nor $\neg\pi$ is provable. If π is a tautology, then so is $p \vee \pi$, which is completely independent of the truth value of p. Since we have no other

information about p, LCW(p) does not follow from anything we know, and thus the query must fail. If neither π nor $\neg\pi$ is provable, then $p \vee \pi$ is irreducible. Since there are some truth assignments under which LCW(p) follows and others under which it doesn't, it's impossible to conclude LCW(p) in general, so again the query must fail.

The above two cases are exhaustive, so we have shown a (linear time) reduction of SAT to LCW inference. Since formula satisfiability is NP-hard, it follows that (unrestricted) LCW inference is also NP-hard.

A.1.2 Proof of Theorem 2.2 (Instantiation)

Let Φ be a logical sentence and suppose LCW(Φ) holds. Let θ be an arbitrary substitution; we need to show that LCW($\Phi\theta$) holds. *I.e.*, by definition of LCW (Equation 2.1) we need show that for all substitutions, σ, either $\mathcal{S} \models \Phi\theta\sigma$ or $\mathcal{S} \models \neg\Phi\theta\sigma$. But since the composition $\theta\sigma$ of substitutions is a substitution, and since LCW(Φ) we conclude LCW($\Phi\theta$).

A.1.3 Proof of Theorem 2.3 (Composition)

Let Φ and Ψ be logical formulas and suppose LCW(Φ) and $\forall\sigma, (\mathcal{S} \not\models \Phi\sigma) \vee$ LCW($\Psi\sigma$). Let θ be an arbitrary substitution. We need to show $[\mathcal{S} \models (\Phi \wedge \Psi)\theta] \vee [\mathcal{S} \models \neg(\Phi \wedge \Psi)\theta]$. If $\mathcal{S} \models (\Phi \wedge \Psi)\theta$, then the proof is complete; so instead assume that $\mathcal{S} \not\models (\Phi \wedge \Psi)\theta$. Since LCW($\Phi$), either $\mathcal{S} \models \Phi\theta$ or $\mathcal{S} \models \neg\Phi\theta$. If $\mathcal{S} \models \neg\Phi\theta$, then clearly $\mathcal{S} \models \neg\Phi\theta \vee \neg\Psi\theta$, and the proof is complete. If $\mathcal{S} \models \Phi\theta$ then $\mathcal{S} \not\models \Psi\theta$ (otherwise, $\mathcal{S} \models (\Phi \wedge \Psi)\theta$). Furthermore, $\mathcal{S} \models \Phi\theta$ implies LCW($\Psi\theta$) (given), so $\mathcal{S} \models \neg\Psi\theta$. Thus $\mathcal{S} \models \neg\Phi\theta \vee \neg\Psi\theta$, which means that LCW($\Phi \wedge \Psi$).

A.1.4 Proof of Theorem 2.4 (Conjunction)

Let Φ and Ψ be logical sentences and suppose LCW(Φ) and LCW(Ψ). By the Instantiation Rule, we have $\forall\sigma$ LCW($\Psi\sigma$), so the condition $\forall\sigma, \mathcal{S} \not\models \Phi\sigma \vee$ LCW($\Psi\sigma$) is trivially true. Thus the Composition Rule applies, and we have LCW($\Phi \wedge \Psi$).

A.1.5 Proof of Theorem 2.5 (Negation)

$\textsc{lcw}(\Phi) \equiv \forall\theta\ [\mathcal{S} \models \Phi\theta] \vee [\mathcal{S} \models \neg\Phi\theta] \equiv \forall\theta\ [\mathcal{S} \models \neg(\neg\Phi\theta)] \vee [\mathcal{S} \models (\neg\Phi\theta)] \equiv \textsc{lcw}(\neg\Phi)$.

A.1.6 Proof of Theorem 2.6 (Disjunction)

By the Theorem 2.5 (Negation), $\textsc{lcw}(\Phi) \wedge \textsc{lcw}(\Psi) \Rightarrow \textsc{lcw}(\neg\Phi) \wedge \textsc{lcw}(neg\Psi)$, which, by Corollary 2.4 (Conjunction), means $\textsc{lcw}(\neg\Phi \wedge \neg\Psi)$, which is equivalent to $\textsc{lcw}(\neg(\Phi \vee \Psi))$, which, by Theorem 2.5 (Negation) means $\textsc{lcw}(\Phi \vee \Psi)$.

A.1.7 Proof of Theorem 2.7 Incompleteness of LCW *Inference Rules*

We provide a simple counter-example. Consider the case in which we know LCW(`in.dir(bak,` f`)` $\wedge$ `is.backup(bak)`), and we also know that `is.backup(bak)` is true. Since `is.backup(bak)` is ground and true, `in.dir(bak,` f`)` $\wedge$ `is.backup(bak)` always has the same truth value as `in.dir(bak,` f`)`. It follows then from the definition of LCW that LCW(`in.dir(bak,` f`)`). Since our inference rules won't derive this formula, they are incomplete. The more general problem is that whenever all possible instances of a formula A are both known and true, LCW(A$\wedge$B) implies LCW(B). For unbounded universes, and a finite knowledge base of positive ground facts, the formula must be ground for all instances to be known true.

A.1.8 Proof of Theorem 2.8 Soundness of `QueryLCW`

We use induction on the number of conjuncts in Φ.

Case $|\Phi| = 0$: An invocation of `QueryLCW` induces a call to QLCW* where line 1 returns `T`. This is correct, because the null clause (*i.e.*, a ground query with zero conjuncts) is unsatisfiable by definition. Since every state in $\mathcal{S}$ agrees that the null clause is false, $\mathcal{S} \models \neg\Phi$ and hence $\textsc{lcw}(\Phi)$.

Case $|\Phi| = k \geq 1$: If QLCW* returns `T`, it must have terminated on line 2 or 6. But line 2 only returns true when all ground instantiations, namely Φ itself, are entailed by $\mathcal{M}$. This corresponds directly to the definition of LCW.

Line 6 will only return `T` under conditions matched by Composition which is sound by Theorem 2.3, or Instantiation (line 4, $\Phi - \Phi' = \{\}$), which is sound by Theorem 2.2. Since these are the only termination points for `QueryLCW`, the algorithm is sound.

A.1.9 Proof of Theorem 2.9 Complexity of `QueryLCW`

Suppose Φ has c conjuncts, let L denote $|\mathcal{L}|$, and let $M = |\mathcal{M}|$. In the worst case, control falls through to line 3, entering a loop over the elements of $\mathcal{L}$, and then in line 4 iterates over at most 2^c conjuncts of Φ. In line 6, the loop body performs a conjunctive match on $\mathcal{M}$ with a pattern whose length is at most $v <= c$ (giving a complexity $O(M^v)$), and then possibly makes a recursive call for each of the $O(M^v)$ possible matches. Thus the following recurrence relation defines the time required by `QueryLCW`:

$$t[c] = L(2^c)(M^v)(t[c-v])$$

Unrolling the recursion yields

$$t[c] = L(2^c)(M^{v_1})(L)(2^{c-v_1})(M^{v_2})\ldots(L)(2^{c-v_1-v_2-\ldots-v_{n-1}})(M^{v_n})$$

The value of t[c] is maximized if each $v_i = 1$. To see why this is, consider that letting $v_i = 1$ maximizes the number of iterations, which maximizes the exponent on the L and the 2. The exponent on the M, on the other hand, doesn't depend on what values are chosen for v_i: since $v_1 + v_2 + \ldots v_n = c$. So the above equation simplifies to

$$t[c] = (L^c)(2^{c^2/2})(M^c)$$

So `QueryLCW` requires at most $O(|\mathcal{L}|^c|\mathcal{M}|^c)$ time. Given that c is bounded by the domain theory, the complexity is polynomial in the size of $\mathcal{L}$ and $\mathcal{M}$.

A.2 Proofs from Chapter 3

A.2.1 Proof of Theorem 3.1 (Successor State Axiom)

Assume $\text{ACHV}(\pi_a, s, \{\})$. We need to show that
$P(\text{DO}(a,s)) \Leftrightarrow \Sigma_P^a(s) \vee (P(s) \wedge \Pi_P^a(s))$ We will first show the "$\Leftarrow$" case: Assume

$$\Sigma_P^a(s) \vee (P(s) \wedge \Pi_P^a(s)) \tag{A.1}$$

By the definition of Σ_P^a (Eqn 3.21),

$$\Sigma_P^a(s) \Rightarrow \gamma_P^{\mathtt{T}}(a,s) \vee (Unk_P(a,s) \wedge \gamma_P^{\mathtt{U}}(a,s)).$$

and by the definition of Π_P^a (Eqn 3.23)

$$\Pi_P^a(s) \Rightarrow \neg\gamma_P^{\mathtt{F}}(a,s) \wedge (Unk_P(a,s) \vee \neg\gamma_P^{\mathtt{U}}(a,s))$$

Applying these two definitions to Eqn A.1 gives us

$$\begin{aligned} &\gamma_P^{\mathtt{T}}(a,s) \vee (Unk_P(a,s) \wedge \gamma_P^{\mathtt{U}}(a,s)) \\ \vee&[P(s) \wedge \neg\gamma_P^{\mathtt{F}}(a,s) \wedge (Unk_P(a,s) \vee \neg\gamma_P^{\mathtt{U}}(a,s))]. \end{aligned} \tag{A.2}$$

As we discussed in Section 3.4.1, because of the Completeness Assumption, **when** clauses are actually bi-implications, so by Equations 3.17–3.19,

$$\begin{aligned} \gamma_P^{\mathtt{T}}(a) &\Leftrightarrow \mathbf{cause}(P,\mathtt{T}) \\ \gamma_P^{\mathtt{F}}(a) &\Leftrightarrow \mathbf{cause}(P,\mathtt{F}) \\ \gamma_P^{\mathtt{U}}(a) &\Leftrightarrow \mathbf{cause}(P,\mathtt{U}) \end{aligned}$$

Applying the above bi-implications to Eqn A.2, we get

$$\begin{aligned} &\text{EFF}(\mathbf{cause}(P,\mathtt{T}),a,s) \vee (Unk_P(a,s) \wedge \text{EFF}(\mathbf{cause}(P,\mathtt{U}),a,s)) \\ \vee&[P(s) \wedge \neg\text{EFF}(\mathbf{cause}(P,\mathtt{F}),a,s) \wedge (Unk_P(a,s) \vee \neg\text{EFF}(\mathbf{cause}(P,\mathtt{U}),a,s))]. \end{aligned} \tag{A.3}$$

By the definitions of $\mathtt{T}$, $\mathtt{F}$ and $\mathtt{U}$ effects (Eqns 3.13–3.15),

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{T}), a, s) \equiv P(\mathrm{DO}(a, s))$$

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{F}), a, s) \equiv \neg P(\mathrm{DO}(a, s))$$

$$\mathrm{EFF}(\mathbf{cause}(P, \mathtt{U}), a, s) \equiv Unk_P(a, \mathrm{DO}(a, s)) \Leftrightarrow P(\mathrm{DO}(a, s))$$

We use $\neg\mathrm{EFF}(\mathbf{cause}(P, \mathtt{F}), a, s)$ in Eqn A.3 as a notational convenience, but we must be careful when interpreting it. $\neg\mathrm{EFF}(\mathbf{cause}(P, \mathtt{F}), a, s)$ simply means that the corresponding update doesn't occur, not that the negation of the update occurs, so propagating the $\neg$ would be an error. So, by Eqn A.3, either there is an update of the form $P(\mathrm{DO}(a, s))$, or $P(s)$ holds and there is no update of the form $\neg P(\mathrm{DO}(a, s))$, which (by the Completeness Assumption discussed in Section 3.1) means $P(\mathrm{DO}(a, s))$.

We will now show the "$\Rightarrow$" case: $P(\mathrm{DO}(a, s))$ holds. There are two possibilities for the value of P in situation s. Either $P(s)$ or $\neg P(s)$. If $P(s)$, then P didn't change truth value, so no effects made it false. There are two effects that could have made it false (Eqns 3.14, 3.15): **cause**(P, $\mathtt{F}$) and **cause**(P, $\mathtt{U}$) (if $\neg Unk_P(a, s)$).

Since neither of these effects occurred, it must be the case that $\neg\gamma_P^{\mathtt{F}}(a, s)$ and either $\neg\mathbf{cause}(P, \mathtt{U})$ (and thus $\neg\gamma_P^{\mathtt{U}}(a, s)$) or $Unk_P(a, s)$.

So by definition (Eqn 3.23), $\Pi_P^a(s)$. If P was false in s then there had to have been an effect that made it true. The two possible effects are (Eqns 3.13, 3.15) **cause**(P, $\mathtt{T}$) and **cause**(P, $\mathtt{U}$) (if $Unk_P(a, s)$). Since one of these actions must have occurred, $\gamma_P^{\mathtt{T}}(a) \vee (Unk_P(a) \wedge \gamma_P^{\mathtt{U}}(a))$, *i.e.*, Σ_P^a.

Putting these two cases together we get $\Sigma_P^a(s) \vee P(s) \wedge \Pi_P^a(s)$, which is what we were trying to prove.

Lemma A.1 (Knowledge effects) *If $K(s', s)$ and not $K(DO(a, s'), DO(a, s))$, then a has an observational effect that results in different observations of some fluent P in s and s'.*

A.2.2 Proof of Lemma A.1 (Knowledge effects)

Assume otherwise: $K(s', s)$ and not $K(\mathrm{DO}(a, s'), \mathrm{DO}(a, s))$, but no observational effect resulted in different observations of any fluent P in s and s'

- If a has *no* effects then, by the Completeness Assumption discussed in Section 3.1, nothing changed between s and s', including K, so $K(s, s') \Leftrightarrow K(\mathrm{DO}(a, s'), \mathrm{DO}(a, s))$ (a contradiction)

- If a has only causal effects, then the contradiction follows from essentially the same argument. The only effects that refer to K are observational effects. Since there were only causal effects, K was unaffected, so by the Completeness Assumption, remains unchanged in situation $\mathrm{DO}(a, s)$, a contradiction.

- So a must have an observational effect that makes s and s' distinguishable. Let that effect be $\mathbf{observe}(P, tv)$. If P is ground, then P must have different truth values in s and s', since otherwise s and s' are not made distinguishable by the observation. If P is not ground (*i.e.*, P contains run-time variables), then there must be some ground instance of P, P', that has different truth values in s and s'.

A.2.3 Proof of Theorem 3.2 (Successor state axiom for K)

Assuming that $\mathrm{ACHV}(\pi_a, s, \{\})$ (*i.e.*, the precondition of a, π_a, is true in situation s), we wish to prove that

$$K(s'', \mathrm{DO}(a, s)) \Leftrightarrow \exists s'.\mathrm{K}(s', s) \wedge (s'' = \mathrm{DO}(a, s')) \wedge \forall P\ (\kappa_P^v(a, s) \Rightarrow [P(s') \Leftrightarrow v])$$

"$\Rightarrow$": Assume otherwise:

$$\exists s''(K(s'', \mathrm{DO}(a, s)) \wedge \forall s'(\neg(\mathrm{K}(s', s) \wedge (s'' = \mathrm{DO}(a, s'))) \vee \exists P\ \kappa_P^v(a, s) \wedge \neg[P(s') \Leftrightarrow v])). \tag{A.4}$$

Applying the definitions of $\kappa_P^v(a,s)$ (Eqn 3.20) and **observe** (Eqn 3.11), we get:

$$\kappa_P^v(a,s) \Rightarrow \forall s'''(\mathrm{K}(s''',\mathrm{DO}(a,s)) \Rightarrow \exists s_i.\mathrm{K}(s_i,s) \wedge s''' = \mathrm{DO}(a,s_i) \wedge (P(s_i) \Leftrightarrow v))$$

Replacing $\kappa_P^v(a,s)$ in Eqn A.4 with the above consequent, and rewriting, we get

$$\begin{aligned} \exists s''\, K(s'',\mathrm{DO}(a,s)) \wedge \forall s'[(\mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s'))) \Rightarrow \\ \exists P\, \forall s'''[\mathrm{K}(s''',\mathrm{DO}(a,s)) \Rightarrow \exists s_i.\mathrm{K}(s_i,s) \wedge s''' = \mathrm{DO}(a,s_i) \wedge (P(s_i) \Leftrightarrow v)] \\ \wedge\neg[P(s') \Leftrightarrow v]] \end{aligned} \quad \text{(A.5)}$$

Since the above holds for all s''', it must hold in particular for $s''' = s''$. Substituting s'' for s''' in Eqn A.5 and simplifying, we get:

$$\begin{aligned} \exists s''\, K(s'',\mathrm{DO}(a,s)) \wedge \forall s'\, (\mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \Rightarrow \exists P\, [\exists s_i.\mathrm{K}(s_i,s) \wedge \\ s'' = \mathrm{DO}(a,s_i) \wedge (P(s_i) \Leftrightarrow v)] \wedge \neg[P(s') \Leftrightarrow v]) \end{aligned} \quad \text{(A.6)}$$

By the Unique Names Assumption (Section 3.1),

$$s'' = DO(a,s') \wedge s'' = DO(a,s_i) \Rightarrow s' = s_i.$$

So we can simplify Eqn A.6 further to

$$\begin{aligned} \exists s''\, K(s'',\mathrm{DO}(a,s)) \wedge \forall s'\, (\mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \Rightarrow \\ \exists P\, [P(s') \Leftrightarrow v] \wedge \neg[P(s') \Leftrightarrow v]) \end{aligned} \quad \text{(A.7)}$$

The consequent of the above implication is false, so the antecedent must be false. Negating it and rewriting it as an implication, we get:

$$\exists s''\, K(s'',\mathrm{DO}(a,s)) \wedge \forall s'\, (\mathrm{K}(s',s) \Rightarrow (s'' \neq \mathrm{DO}(a,s')))$$

But this says that it's consistent with the agent's knowledge in situation $\mathrm{DO}(a,s)$, but *inconsistent* with the agent's knowledge in s, to believe that the actual situation

after executing a is s'', meaning the set of possible situations increased after executing a, which cannot happen — a contradiction.

"$\Leftarrow$": We will assume

$$\neg \mathrm{K}(s'', \mathrm{DO}(a,s)) \wedge \exists s' \, \mathrm{K}(s', s) \wedge (s'' = \mathrm{DO}(a, s')) \wedge \\ \forall P \, (\kappa_P^v(a,s) \Rightarrow [P(s') \Leftrightarrow v]) \tag{A.8}$$

and show that we get a contradiction. By Eqn A.8, $\exists s'(\neg \mathrm{K}(\mathrm{DO}(a,s'), \mathrm{DO}(a,s)) \wedge \mathrm{K}(s', s))$, *i.e.*, the agent has information in situation $\mathrm{DO}(a,s)$ that it didn't have in situation s. By Lemma A.1, a must have an observational effect in situation s that results in different observations of some fluent φ in s and s'. By the definition of **observe** (Eqn 3.11), the observed truth value of φ in situation s is v, so $\varphi(s')$ must have a different truth value than v. By Eqn A.8, $\kappa_\varphi^v(a,s) \Rightarrow [\varphi(s') \Leftrightarrow v]$. *i.e.*, φ has the same truth value as v, a contradiction. Thus the assumption must be wrong, and the implication holds.

A.2.4 Proof of Theorem 3.5 (No Correlations)

Assume there are no correlations between unknown plan fluents in situation s_0, no disjunctive effects, and that no action a_i in $\{a\}_1^n$, or any effect of a_i, has any preconditions that are unknown in the situation $\mathrm{DO}(a_i, s_i)$. We will prove that there will be no correlations between unknown plan fluents at any time during the execution of $\{a\}_1^n$.

We will prove the theorem for the execution of a single action. It follows by induction on the number of actions that the result holds for the whole plan.

Assume otherwise. There must have been some effects in action a that introduced correlations between unknown fluents, despite the previous lack of correlations. The only possible effects are ones with preconditions of the form $\gamma_\varphi^{\mathtt{T}}(a)$, $\gamma_\varphi^{\mathtt{F}}(a)$, $\gamma_\varphi^{\mathtt{U}}(a)$ or $\kappa_\varphi^{tv}(a)$. Since, by assumption the preconditions are known after execution of a, the effects of `T` and `F` updates must also be known afterward. Similarly, effects with

$\kappa^{tv}_{\varphi}(a)$ preconditions only result in φ being known. Such an effect can't result in correlations between unknown fluents because it makes fluent φ known, depends only on known preconditions, and influences only the single fluent φ (with no disjunction), which, by the induction hypothesis, can't be previously correlated with other fluents. An effect with a $\gamma^{\mathtt{U}}_{\varphi}(a)$ precondition can make a fluent φ unknown, but depends causally only on $\gamma^{\mathtt{U}}_{\varphi}(a)$, which is known, and Unk_{φ}. By definition, Unk_{φ} must be unknown. Since Unk_{φ} can't be listed in preconditions or effects of any action, and since by assumption it is uncorrelated with any unknown fluents, it remains uncorrelated with everything except φ. After execution, φ is correlated with Unk_{φ}, but since Unk_{φ} cannot be mentioned anywhere in the plan, there is no way, through Unk_{φ}, to introduce correlations between φ and other fluents. So φ, though unknown, is uncorrelated with all other fluents in the plan.

A.2.5 Proof of Theorem 3.6 (Successor state axiom for KNOW*)*

We wish to show that

$$\text{KNOW}(P, DO(a,s)) \Leftrightarrow \Sigma^a_{\text{KNOW}(P)}(s) \vee (\text{KNOW}(P,s) \wedge \Pi^a_{\text{KNOW}(P)}(s)).$$

"$\Leftarrow$": Assume $\Sigma^a_{\text{KNOW}(P)}(s) \vee (\text{KNOW}(P,s) \wedge \Pi^a_{\text{KNOW}(P)}(s))$. One of the following cases holds:

1. $\Sigma^a_{\text{KNOW}(P)}(s)$. By definition (Eqn. 3.26),

   $\Sigma^a_{\text{KNOW}(P)}(\text{s}) \Rightarrow \text{KNOW}(\gamma^{\mathtt{T}}_P(a,s)) \vee (\kappa^{tv}_{\gamma^{\mathtt{T}}_P(a)}(a,s) \wedge \gamma^{\mathtt{T}}_P(a,s)) \vee$
   $(\kappa^{\mathtt{T}}_P(a,s) \wedge P(s) \wedge \Pi^a_P(s))$.

   So one of the following holds: $\text{KNOW}(\gamma^{\mathtt{T}}_P(a,s))$, or $(\kappa^{tv}_{\gamma^{\mathtt{T}}_P(a)}(a,s) \wedge \gamma^{\mathtt{T}}_P(a,s))$, or $(\kappa^{\mathtt{T}}_P(a,s) \wedge P(s) \wedge \Pi^a_P(s))$. We consider each in turn:

   (a) $\text{KNOW}(\gamma^{\mathtt{T}}_P(a,s))$. By Eqn 3.21, this means $\text{KNOW}(\Sigma^a_P(s))$, which (by Theorem 3.1 (SSA)) gives us $\text{KNOW}(P(DO(a,s)))$.

(b) $\kappa_P^{\mathtt{T}}(a,s) \wedge P(s) \wedge \Pi^a_{\mathrm{KNOW}(P)}(s)$. By Theorem 3.2 (SSA for K),

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow \exists s' \, \mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \wedge \forall \varphi (\kappa_P^v(a,s) \Rightarrow [\varphi(s') \Leftrightarrow v])] \tag{A.9}$$

Letting $\varphi = P$, and exploiting the assumption that $\kappa_P^{\mathtt{T}}(a,s)$, we get

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow \exists s' \, \mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \wedge P(s') \Leftrightarrow v] \tag{A.10}$$

Letting $s'' = \mathrm{DO}(a,s)$ in Eqn A.10 gives us

$$\forall s''[K(\mathrm{DO}(a,s), \mathrm{DO}(a,s)) \Rightarrow (P(s) \Leftrightarrow v)]$$

But by definition, $\forall s^*(\mathrm{K}(s^*, s^*))$, so the antecedent above must be true, meaning $(P(s) \Leftrightarrow v)$. Since $P(s)$ is true by assumption, v must also be true. Applying this observation to Eqn A.10, we get

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow \exists s' \, \mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \wedge P(s')] \tag{A.11}$$

$\Pi^a_{\mathrm{KNOW}(P)}(s)$, by assumption, which, by definition, is equivalent to $\mathrm{KNOW}(\Pi_P^{a,s})$, meaning $\mathrm{K}(s',s) \Rightarrow \Pi_P^a(s')$. Since $P(s')$ and $\Pi_P^{a,s}$, by the successor state axiom (Theorem 3.1) for P in situation s', it follows that $P(\mathrm{DO}(a,s'))$. Applying this to Eqn A.11 and simplifying gives us

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow P(\mathrm{DO}(a,s')) \tag{A.12}$$

i.e., $\mathrm{KNOW}(P, \mathrm{DO}(a,s))$.

(c) $\kappa^{tv}_{\gamma_P^{\mathtt{T}}(a,s)}(a) \wedge \gamma_P^{\mathtt{T}}(a,s)$. By Theorem 3.2 (SSA for K),

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow \exists s' \, \mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \wedge \forall \varphi (\kappa_P^v(a,s) \Rightarrow [\varphi(s') \Leftrightarrow v])] \tag{A.13}$$

Let $\varphi = \gamma_P^{\mathtt{T}}(a)$. By the argument given in item (c) and elsewhere, $\gamma_P^{\mathtt{T}}(a,s) \Leftrightarrow v$. Since $\gamma_P^{\mathtt{T}}(a,s)$ is true by assumption, so is v. Also by assumption, $\kappa^{tv}_{\gamma_P^{\mathtt{T}}(a,s)}(a)$. Given these facts, Eqn A.13 reduces to

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow \exists s' \, \mathrm{K}(s',s) \wedge (s'' = \mathrm{DO}(a,s')) \wedge \gamma_P^{\mathtt{T}}(a,s') \quad \text{(A.14)}$$

By definition (Eqn 3.21),$\gamma_P^{\mathtt{T}}(a)(s') \Rightarrow \Sigma_P^a(s')$. By By Theorem 3.1 (SSA for P), $\Sigma_P^a(s') \Rightarrow P(\mathrm{DO}(a,s'))$. Thus, Eqn A.14 can be simplified to:

$$\forall s'' \, [K(s'', \mathrm{DO}(a,s)) \Rightarrow P(s'')]$$

Which, by definition, is $\mathrm{KNOW}(P(DO(a,s)))$.

2. $\mathrm{KNOW}(P,s) \wedge \Pi^a_{\mathrm{KNOW}(P)}(s)$. By definition (Eqn 3.25), $\Pi^a_{\mathrm{KNOW}(P)}(s) \Rightarrow \mathrm{KNOW}(\Pi^a_P, s)$. Applying the definition of KNOW, $\mathrm{KNOW}(P) \wedge \mathrm{KNOW}(\Pi^a_P, s)$ gives us

$$\forall s' (K(s',s) \Rightarrow P(s') \wedge \Pi^a_P(s'))$$

By Theorem 3.1 (SSA) applied to P in situation s', this gives us

$$\forall s' (K(s',s) \Rightarrow P(\mathrm{DO}(a,s')))$$

By Theorem 3.2 (SSA for K),

$$\forall s'' (\mathrm{K}(s'', \mathrm{DO}(a,s)) \Rightarrow K(s',s) \wedge (s'' = \mathrm{DO}(a,s')))$$

Combining the above two equations, we get

$$\forall s'' (\mathrm{K}(s'', \mathrm{DO}(a,s)) \Rightarrow P(s''))$$

which is equivalent to $\mathrm{KNOW}(P, DO(a,s))$.

"$\Rightarrow$": Assume $\mathrm{KNOW}(P, DO(a,s))$. We want to show

$$\Sigma^a_{\mathrm{KNOW}(P)}(s) \vee (\mathrm{KNOW}(P,s) \wedge \Pi^a_{\mathrm{KNOW}(P)}(s)) \quad \text{(A.15)}$$

First, we will show that $\Pi^a_{\text{KNOW}(P)}(s)$. Assume otherwise. By definition (Eqn 3.25),

$$\neg\Pi^a_{\text{KNOW}(P)}(s) \Rightarrow \neg\text{KNOW}(\Pi^a_P, s).$$

By the definition of KNOW, we can rewrite this as

$$\exists s'(\text{K}(s', s) \wedge \neg\Pi^a_P(s')).$$

But by the definitions of Π^a_P (Eqn 3.23) and Σ^a_P (Eqn 3.21), $\neg\Pi^a_P(s') \Rightarrow \Sigma^a_{\neg P}(s')$, and by Theorem 3.1 (SSA), $\Sigma^a_{\neg P}(s') \Rightarrow \neg P(\text{DO}(a, s')$, so we have

$$\exists s'(\text{K}(s', s) \wedge \neg P(\text{DO}(a, s'))),$$

which is equivalent to $\neg\text{KNOW}(P, \text{DO}(a, s))$, a contradiction. Therefore $\Pi^a_{\text{KNOW}(P)}(s)$.

Now either $\text{KNOW}(P, s)$ or $\neg\text{KNOW}(P, s)$. We will consider each case in turn.

- If $\text{KNOW}(P, s)$, then since $\Pi^a_{\text{KNOW}(P)}$, Eqn A.15 is satisfied, and we're done.
- Assume $\neg\text{KNOW}(P, s)$. But P is known in situation DO(a, s), possibly because P changed. We will consider two exhaustive cases: Either executing a in s is known to make P true, or it isn't.

  1. The action is known to make P true: $\forall s'\ \text{K}(s', s) \Rightarrow P(\text{DO}(a, s'))$. By Theorem 3.1 (SSA), we can replace $P(\text{DO}(a, s'))$, giving us

     $$\forall s'\ \text{K}(s', s) \Rightarrow P(\text{DO}(a, s')) \Rightarrow \Sigma^a_P(s') \vee (P(s') \wedge \Pi^a_P(s')).$$

     Applying the definition of Σ^a_P (Eqn 3.21), this gives us

     $$\forall s'\ \text{K}(s', s) \Rightarrow \gamma^{\mathtt{T}}_P(a)(s') \vee (Unk_P(a, s') \wedge \gamma^{\mathtt{U}}_P(a)(s')) \vee (P(s') \wedge \Pi^a_P(s')),$$

     which is equivalent to $\text{KNOW}(\gamma^{\mathtt{T}}_P(a) \vee (Unk_P(a) \wedge \gamma^{\mathtt{U}}_P(a)) \vee (P \wedge \Pi^a_P), s)$. We want to conclude $\text{KNOW}(\gamma^{\mathtt{T}}_P(a), s)$, so we will argue away the other terms.

$\gamma_P^{\mathtt{U}}(a)(s')$ must be false for all situations s' such that $K(s', s)$. The reason is that $Unk_P(a)$ cannot, by definition be correlated with $\gamma_P^{\mathtt{U}}(a)$. Hence if $\gamma_P^{\mathtt{U}}(a)$ is true in some situations, $Unk_P(a)$ will be false in some of those, resulting in P becoming false, contradicting our assumption that the action is known to make P true. Or more succinctly, U effects make a condition unknown.

This leaves us with $\mathrm{KNOW}(\gamma_P^{\mathtt{T}}(a) \vee (P \wedge \Pi_P^a), s)$. By Theorem 3.5, there are no correlations in the agent's knowledge, so $\mathrm{KNOW}(\gamma_P^{\mathtt{T}}(a)) \vee \mathrm{KNOW}(P \wedge \Pi_P^a, s)$. We know that $\neg\mathrm{KNOW}(P, s)$, by assumption, so $\mathrm{KNOW}(\gamma_P^{\mathtt{T}}(a))$, which implies $\Sigma^a_{\mathrm{KNOW}(P)}$.

2. The action a isn't known in s to make P true: $\exists s'.\mathrm{K}(s', s)) \wedge \neg P(\mathrm{DO}(a, s'))$. Let us pick one such s'. But $\mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow P(\mathrm{DO}(a, s'))$, so $\neg\mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s))$. That is, executing a reduces the set of possible worlds consistent with the agent's knowledge. By Theorem 3.2 (SSA for K), it follows that there is some predicate φ such that $\kappa_\varphi^v(a, s) \wedge \neg(\varphi(s') \Leftrightarrow v)$; *i.e.*, some predicate φ must have been sensed to eliminate $\mathrm{DO}(a, s')$ from the set of possible worlds. We cannot assume that $\varphi = P$, but we'll consider both possibilities.

   (a) Assume $\kappa_P^v(a, s)$. If $P(s)$ then since $\Pi^a_{\mathrm{KNOW}(P)}(s)$, by the definition of $\Sigma^a_{\mathrm{KNOW}(P)}$ (Eqn 3.26), $\kappa_P^v(a, s) \wedge P(s) \wedge \Pi^a_{\mathrm{KNOW}(P)}(s) \Rightarrow \Sigma^a_{\mathrm{KNOW}(P)}$, and Eqn A.15 is satisfied, so assume otherwise: $\kappa_P^v(a, s) \wedge \neg P(s)$, By Theorem 3.2 (SSA for K), we have $\mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow \kappa_P^v(a, s) \Rightarrow (P(s') \Leftrightarrow v)$. By the same argument given in part 1(b) of this proof, $P(s) \Leftrightarrow v$, so we can conclude $\neg v$. Thus, $\mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow \neg P(s')$, *i.e.*, since in

situation s, P was observed (and false), it is known in $\mathrm{DO}(a,s)$ that $\neg P(s)$. But we know that P is true in situation $\mathrm{DO}(a,s)$, so it must have become true. By Theorem 3.1 (SSA applied to P in situation s'), $\neg P(s') \wedge P(\mathrm{DO}(a,s')) \Rightarrow \Sigma^a_P(s')$, which, by the definition of Σ^a_P, gives $\gamma^{\mathtt{T}}_P(a)(s') \vee (Unk_P(a,s') \wedge \gamma^{\mathtt{U}}_P(a)(s'))$.

By the argument from case 1, we can eliminate the $\gamma^{\mathtt{U}}_P(a)$ term, leaving us with $\mathrm{K}(\mathrm{DO}(a,s'),\mathrm{DO}(a,s)) \Rightarrow \gamma^{\mathtt{T}}_P(a)(s')$, *i.e.*, it is known after a is executed in situation s that $\gamma^{\mathtt{T}}_P(a)$ was true in s. This fact was not known in s, by assumption, so it became known after executing a (becoming *true* isn't sufficient, since the condition needs to have held *before* a was executed):

$\kappa^{tv}_{\gamma^{\mathtt{T}}_P(a)}(a,s) \wedge \gamma^{\mathtt{T}}_P(a,s) \Rightarrow \Sigma^a_{\mathrm{KNOW}(P)}$.

(b) if $\neg\kappa^v_P(a,s)$ then some other fluent(s) must have been sensed. Let ψ be the conjunction of all fluents that were sensed. By Theorem 3.2 (SSA for K), we have

$$\begin{aligned}\forall s''\ \mathrm{K}(s'',\mathrm{DO}(a,s)) \Leftrightarrow \exists s'.K(s',s)\\ \wedge(s''=\mathrm{DO}(a,s')) \wedge (\psi(s') \Leftrightarrow v),\end{aligned} \tag{A.16}$$

i.e., the only information gained was about ψ. Yet we know that $\mathrm{K}(\mathrm{DO}(a,s'),\mathrm{DO}(a,s)) \Rightarrow P(\mathrm{DO}(a,s'))$. There are two possibilities: Either $\forall s^*(\mathrm{K}(\mathrm{DO}(a,s^*),\mathrm{DO}(a,s)) \Rightarrow P(s^*))$, *i.e.*, after executing a, the agent knows that P was true in s, or the agent has no such knowledge: $\exists s^*(\mathrm{K}(\mathrm{DO}(a,s^*),\mathrm{DO}(a,s)) \wedge \neg P(s^*))$.

- Assume $\forall s^*(\mathrm{K}(\mathrm{DO}(a,s^*),\mathrm{DO}(a,s)) \Rightarrow P(s^*)$. Letting $s^*=s'$, we can combine this with Eqn A.16, exploiting the $\Leftrightarrow$, to get:

$$\begin{aligned}\forall s''\exists s'\ \mathrm{K}(s'',\mathrm{DO}(a,s)) \Rightarrow \mathrm{K}(s',s)\\ \wedge(s''=\mathrm{DO}(a,s')) \wedge (\psi(s') \Leftrightarrow v) \Rightarrow P(s').\end{aligned} \tag{A.17}$$

Since $\forall s'\ \mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow \mathrm{K}(s', s)$, we can eliminate the $\mathrm{K}(s', s)$ term and simplify. We can eliminate the s'' and change the $\exists s'$ to a $\forall s'$ by exploiting the Unique Names Assumption, which says that there is only one s' such that $s'' = \mathrm{DO}(a, s')$:

$$\forall s'\ \mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow (\psi(s') \Leftrightarrow v) \Rightarrow P(s'). \quad \text{(A.18)}$$

Depending on the value of v, this gives us either
$\forall s'\ \mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow (\psi(s') \Rightarrow P(s'))$ or
$\forall s'\ \mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow (\neg\psi(s') \Rightarrow P(s'))$.
The agent knows after executing a that the value of P(s) is correlated with the value of $\psi(s)$. Since P was unknown in s, by Theorem 3.5, the agent cannot have known about this correlation in situation s, so it must have discovered it after executing a. But since effects cannot contain disjunction or implication, there's no way the agent could have made this discovery without independently discovering the value of $P(s)$ and $\psi(s)$. By assumption, the agent did not directly observe the value of P — a contradiction. Therefore:

– $\exists s^*(\mathrm{K}(\mathrm{DO}(a, s^*), \mathrm{DO}(a, s)) \wedge \neg P(s^*)$, but $\forall \mathrm{K}(\mathrm{DO}(a, s'), \mathrm{DO}(a, s)) \Rightarrow P(\mathrm{DO}(a, s'))$, *i.e.*, it is known that P is true after execution. Since P is true in situation $\mathrm{DO}(a, s^*)$ but false in situation s^*, it must have become true: $\neg P(s^*) \wedge P(\mathrm{DO}(a, s^*)) \Rightarrow \Sigma^a_P(s^*)$. By the argument in case 2a, $\gamma^{\mathtt{T}}_P(a)(s^*)$. So either $\mathrm{KNOW}(\gamma^{\mathtt{T}}_P(a, s))$, or it is unknown in s whether or not $\gamma^{\mathtt{T}}_P(a)(s)$. By assumption, the truth value of all preconditions must be known either just prior to execution or immediately after, so by definition (Eqn 3.26), $\mathrm{KNOW}(\gamma^{\mathtt{T}}_P(a, s)) \vee \kappa^{tv}_{\gamma^{\mathtt{T}}_P(a)}(a, s) \wedge \gamma^{\mathtt{T}}_P(a, s) \Rightarrow \Sigma^a_{\mathrm{KNOW}(P)}$.

A.2.6 Proof of Theorem 3.7 (Soundness of regression)

We wish to show that $\alpha \models (\text{ACHV}(\mathsf{R}_{\{a\}_1^n}(\Gamma), s_0, \{\}) \Rightarrow \text{ACHV}(\Gamma, s_0, \{a\}_1^n))$ The proof is by induction on the number of actions in $\{a\}_1^n$.

Base case: $n = 0$. By the definition of $\mathsf{R}_{\{\}}$, $\alpha \models \mathsf{R}_{\{a\}_1^n}(\Gamma)$ iff $\alpha \models \Gamma$. Equations 3.27 and 3.29 give the conditions under which $\alpha \models \Gamma$ for **initially**, **satisfy** and **hands-off** goals. These conditions conform directly to the definitions of **initially**, **satisfy** and **hands-off** given in equations 3.3 – 3.5, for the case $n = 0$.

If $n > 0$, then we have the following cases.

1. $\mathsf{R}_{\{a\}_1^n}(\textbf{satisfy}(\text{P})) \Rightarrow \Sigma^a_{\text{KNOW}(P)} \vee \textbf{satisfy}(\varphi) \wedge \Pi^a_{\text{KNOW}(\varphi)}$, by definition (Equation 3.30). Applying the definition of **satisfy**(φ) (Equation 3.3) yields: $\Sigma^a_{\text{KNOW}(\varphi)}(s_{n-1}) \vee \text{KNOW}(\varphi, s_{n-1}) \wedge \Pi^a_{\text{KNOW}(\varphi)}(s_{n-1})$ By Theorem 3.6 (SSA for KNOW), this implies $\text{KNOW}(\varphi, s_n)$, which, by definition (Eqn 3.3), means $\text{ACHV}(\textbf{satisfy}(\varphi), s_0, \{a\}_1^n)$. There is no need to invoke the induction hypothesis for **satisfy**.

2. $\mathsf{R}_{\{a\}_1^n}(\textbf{hands-off}(\varphi)) =$
   $(\text{KNOW}(\Pi^a_{\neg\varphi}) \vee \textbf{initially}(\varphi)) \wedge (\text{KNOW}(\Pi^a_{\varphi}) \vee \textbf{initially}(\neg\varphi)) \wedge \textbf{hands-off}(\varphi)$.
   By assupmtion, this formula regresses sucessfully back to the initial state $(\text{ACHV}(\mathsf{R}_{\{a\}_1^n}(\Gamma), s_0, \{\}))$, so the regression of each conjunct must be true. By the induction hypothesis, $\text{ACHV}(\mathsf{R}_{\{a\}_1^{n-1}}(\textbf{hands-off}(\varphi)), s_0, \{\}) \Rightarrow$ $\text{ACHV}((\textbf{hands-off}(\varphi)), s_0, \{a\}_1^{n-1})$. But by definition of **hands-off**(Eqn 3.5),

   $\text{ACHV}((\textbf{hands-off}(\varphi)), s_0, \{a\}_1^{n-1}) \Rightarrow \bigwedge_{i=1}^{n-1} \forall s \in \text{ORIG}_i\ \varphi(\text{DO}(\{a\}_1^i, s) \Leftrightarrow \varphi(s))$, so we need merely show that $\forall s \in \text{ORIG}_n\ [\varphi(\text{DO}(\{a\}_1^n, s) \Leftrightarrow \varphi(s))]$ to show that $\text{ACHV}(\textbf{find-out}(\varphi), s_0, \{a\}_1^n)$.

   Following the same reasoning as above, we can apply the induction hypothesis to the **initially** preconditions, giving:

$([\forall s \in \text{ORIG}_{n-1}\, \varphi(s)] \vee\ \text{KNOW}(\Pi^{a}_{\neg\varphi}), s_{n-1}) \wedge ([\forall s \in$
$\text{ORIG}_{n-1}\, \neg\varphi(s)] \vee \text{KNOW}(\Pi^{a}_{\varphi}), s_{n-1})$

This can be rewritten as:

$\forall s \in \text{ORIG}_{n-1}\, [\varphi(s) \vee\ \Pi^{a}_{\neg\varphi}(\text{DO}(\{a\}_1^{i-1}, s))] \wedge \forall s \in$
$\text{ORIG}_{n-1}\, [\neg\varphi(s) \vee \Pi^{a}_{\varphi}(\text{DO}(\{a\}_1^{i-1}, s))]$

Consider the following (exhaustive) cases:

(a) If $\varphi(s)$ is unknown in s_{n-1}, then this formula simplifies to

$\forall s \in \text{ORIG}_{n-1}\, [\Pi^{a}_{\neg\varphi}(\text{DO}(\{a\}_1^{i-1}, s)) \wedge \Pi^{a}_{\varphi}(\text{DO}(\{a\}_1^{i-1}, s))]$

By Theorem 3.1 (SSA),

$(\forall s \in \text{ORIG}_{n-1}\, [\varphi(\text{DO}(\{a\}_1^{n}, s)) \Leftrightarrow \varphi(\text{DO}(\{a\}_1^{n}, s)))]$

Since ORIG_{n-1} is a superset of ORIG_{n} and since we already have, through the induction hypothesis, $\varphi(\text{DO}(\{a\}_1^{n-1}, s) \Leftrightarrow \varphi(s))$, this implies $(\forall s \in \text{ORIG}_{n}\, [\varphi(\text{DO}(\{a\}_1^{n}, s)) \Leftrightarrow \varphi(s))]$, which is what we wanted to prove.

(b) If the value of $\varphi(s)$ is known in s_{n-1}, then either it is known true or known false. Assume, WLOG, that it is known true (the argument for the opposite case is identical).

Then $\forall s \in \text{ORIG}_{n-1}\, [\varphi(s) \wedge \Pi^{a}_{\varphi}(\text{DO}(\{a\}_1^{i-1}, s))]$.

We already know, by the induction hypothesis, that $\varphi(s) \Rightarrow \varphi(\text{DO}(\{a\}_1^{i-1}, s))$. By Theorem 3.1 (SSA),

$\forall s \in \text{ORIG}_{n-1}\, [\varphi(s) \wedge \varphi(\text{DO}(\{a\}_1^{n}, s))]$. which is what we wanted to prove.

3. $\mathsf{R}_{\{a\}_1^n}(\textbf{initially}(\varphi)) \Rightarrow \textbf{initially}(\varphi) \vee (\kappa^{tv}_{\varphi}(a) \wedge \varphi \wedge \textbf{hands-off}(\varphi))$

(a) If $\textbf{initially}(\varphi,\, s_{n-1})$, then by the induction hypothesis, $\text{ACHV}(\textbf{initially}(\varphi), s_0, \{a\}_1^{n-1}) \Rightarrow \forall s \in \text{ORIG}_n\, [\varphi(s)]$. Applying

Theorem 3.2 (SSA for K) to ORIG reveals that $\mathrm{ORIG}_n \subseteq \mathrm{ORIG}_{n-1}$, from which it follows, by definition, that $\mathrm{ACHV}(\mathbf{initially}(\varphi), s_0, \{a\}_1^n)$.

(b) Otherwise, by the induction hypothesis, applied to **hands-off**, φ is known to have the same truth value in situations s_0 and s_{n-1}: $\forall s \in \mathrm{ORIG}_{n-1}\,[\varphi(\mathrm{DO}(\{a\}_1^{n-1}, s)) \Leftrightarrow \varphi(s)]$. Furthermore, because $\kappa_\varphi^{tv}(a)(s_{n-1}) \wedge \varphi(s_{n-1})$, Theorem 3.2 dictates that φ is known in s_n to have been true in s_{n-1}: $\forall s \in \mathrm{ORIG}_n\,[\varphi(\mathrm{DO}(\{a\}_1^{n-1}, s))]$ Combining these results, we get $\forall s \in \mathrm{ORIG}_n\ \varphi(s)$, so $\mathrm{ACHV}(\mathbf{initially}(\varphi), s_0, \{a\}_1^n)$.

4. $\mathsf{R}_{\{a\}_1^n}(\varphi) \Rightarrow \Sigma_\varphi^a(s) \vee \varphi(s) \wedge \Pi_\varphi^a(s) \Rightarrow \varphi(\mathrm{DO}(a, s))$, by definition (Eqn 3.33), but that also satisfies the successor state axiom for φ(Theorem 3.1), so, by definition, $\mathrm{ACHV}(\varphi, s_0, \{a\}_1^n)$.

A.2.7 Proof of Theorem 3.8 (Completeness of regression)

The proof is by induction on the number of actions in $\{a\}_1^n$. The base case, $n = 0$, is discussed in the proof for Theorem 3.7.

If $n > 0$, then we have the following cases.

1. $\mathrm{ACHV}(\mathbf{satisfy}(\varphi), s_0, \{a\}_1^n) \Rightarrow \mathrm{KNOW}(\varphi(\mathrm{DO}(a, s_{n-1})))$.

   By Theorem 3.6 (SSA for KNOW), $\mathrm{KNOW}(\varphi, \mathrm{DO}(a, s_n)) \Rightarrow$ $\Sigma^a_{\mathrm{KNOW}(\varphi)}(s_{n-1}) \vee \mathrm{KNOW}(\varphi, s_{n-1}) \wedge \Pi^a_{\mathrm{KNOW}(\varphi)}(s_{n-1})$. But by the definition of **satisfy**, we can rewrite that as $\Sigma^a_{\mathrm{KNOW}(\varphi)}(s_{n-1}) \vee (\mathbf{satisfy}(\varphi, (s_{n-1})) \wedge \Pi^a_{\mathrm{KNOW}(\varphi)}(s_{n-1}))$, which, by definition of the regression operator for **satisfy**, gives us $\mathsf{R}_{\{a\}_1^n}(\mathbf{satisfy}(\varphi))$. As with the previous theorem, we don't need the induction hypothesis for **satisfy**.

2. $\mathrm{ACHV}(\mathbf{hands\text{-}off}(\varphi), s_0, \{a\}_1^n) \Rightarrow \forall s \in \mathrm{ORIG}_n\,[\varphi(\mathrm{DO}(\{a\}_1^{n-1}, s)) \Leftrightarrow \varphi(s)] \wedge \bigwedge_{i=1}^{n-1} \forall s \in \mathrm{ORIG}_i\,[\varphi(\mathrm{DO}(\{a\}_1^i, s) \Leftrightarrow \varphi(s)]$, by definition (Equation 3.5).

The second conjunct, $\bigwedge_{i=1}^{n-1} \forall s \in \mathrm{ORIG}_i\, [\varphi(\mathrm{DO}(\{a\}_1^i, s) \Leftrightarrow \varphi(s))]$, satisfies the definition of ACHV(**hands-off**, $s_0, \{a\}_1^{n-1}$). By the induction hypothesis, $\mathsf{R}_{\{a\}_1^{n-1}}(\textbf{hands-off}(\varphi))$.

The first conjunct, $\forall s \in \mathrm{ORIG}_n\, [\varphi(\mathrm{DO}(\{a\}_1^{n-1}, s)) \Leftrightarrow \varphi(s)]$, yields, by Theorem 3.1 (SSA applied to φ),
$\forall s \in \mathrm{ORIG}_n\; \Sigma_\varphi^a(s_{n-1}) \vee \varphi(s_{n-1}) \wedge \Pi_\varphi^a(s_{n-1}) \Leftrightarrow \varphi(s)$, Since by assumption, **hands-off**(φ) holds in s_n, by the definition of **hands-off**, $\varphi(s_{n-1}) \Leftrightarrow \varphi(s_0)$, so this simplifies and expands to $\forall s \in \mathrm{ORIG}_n\;\; [\Sigma_\varphi^a(s) \vee\, \Pi_\varphi^a(s)] \wedge \varphi(s_0) \vee$
$[\Sigma_{\neg\varphi}^a(s) \vee\, \Pi_{\neg\varphi}^a(s)] \wedge \neg\varphi(s_0)$.

Note that the rules of consistent effects imply $\Sigma_\varphi^a \Rightarrow \neg\Sigma_{\neg\varphi}^a \Rightarrow \Pi_\varphi^a$, and similarly for $\Sigma_{\neg\varphi}^a$. So, $\forall s \in \mathrm{ORIG}_n\;\; [\Pi_\varphi^a(s) \vee\, \Pi_\varphi^a(s)] \wedge \varphi(s_0) \vee\, [\Pi_{\neg\varphi}^a(s) \vee$
$\Pi_{\neg\varphi}^a(s)] \wedge \neg\varphi(s_0)$. Which simplifies to $\forall s \in \mathrm{ORIG}_n\;\; [(\Pi_\varphi^a(s) \wedge \varphi(s_0)) \vee$
$(\Pi_{\neg\varphi}^a(s) \wedge \neg\varphi(s_0))]$.

Converting to POS form, $\forall s \in \mathrm{ORIG}_n\;\; [\Pi_\varphi^a(s) \vee \Pi_{\neg\varphi}^a(s)]\; \wedge [\Pi_\varphi^a(s) \vee \neg\varphi(s_0))] \wedge$
$[\varphi(s_0) \vee \Pi_{\neg\varphi}^a(s)] \wedge\, [\varphi(s_0) \vee \neg\varphi(s_0)]$. The first term is implied by the requirement that effects be consistent. The last term is a tautology. Dropping these terms, we get
$\forall s \in \mathrm{ORIG}_n\; ([\neg\varphi(s_0) \vee \Pi_\varphi^a(s)] \wedge [\varphi(s_0) \vee \Pi_{\neg\varphi}^a(s)])$

Theorem 3.5 dictates that there are no correlated unknown truth values, meaning $\mathrm{KNOW}(A \vee B) \Rightarrow \mathrm{KNOW}(A) \vee \mathrm{KNOW}(B)$, so we can rewrite:
$[(\forall s \in \mathrm{ORIG}_n\; \neg\varphi(s_0)) \vee (\forall s \in \mathrm{ORIG}_n\; \Pi_\varphi^a(s))] \wedge [(\forall s \in \mathrm{ORIG}_n\; \varphi(s_0)) \vee (\forall s \in$
$\mathrm{ORIG}_n\; \Pi_{\neg\varphi}^a(s))]$. By the definition of **initially** and the induction hypothesis,
$[\textbf{initially}(\varphi) \vee \mathrm{KNOW}(\Pi_\varphi^a, s_{n-1})] \wedge [\textbf{initially}(\neg\varphi) \vee \mathrm{KNOW}(\Pi_{\neg\varphi}^a, s_{n-1})]$.
Which implies $\mathsf{R}_{\{a\}_1^n}(\textbf{hands-off}(\varphi))$.

3. ACHV(**initially**(φ), $s_0, \{a\}_1^n$) $\Rightarrow \forall s \in \mathrm{ORIG}_n\, [\varphi(s)]$. Either:

- $\forall s \in \text{ORIG}_{n-1}\, \varphi(s)$. By the induction hypothesis, $\mathsf{R}_{\{a\}_1^{n-1}}(\textbf{initially}(\varphi))$, which satisfies the definition of $\mathsf{R}_{\{a\}_1^{n}}(\textbf{initially}(\varphi))$.
- Or $\exists \in \text{ORIG}_{n-1}\, \neg\varphi(s)$. But since $\forall s \in \text{ORIG}_n\, [\varphi(s)]$, (applying definitions of ORIG_{n-1} and ORIG_n) $\exists s$ such that $\text{K}(\text{DO}(\{a\}_1^{n-1}, s), \text{DO}(\{a\}_1^{n-1}, s_0)) \wedge \neg\varphi(s) \wedge [\text{K}(\text{DO}(\{a\}_1^{n}, s), \text{DO}(\{a\}_1^{n}, s_0)) \Rightarrow \varphi(s)]$. Since $\neg\varphi(s)$, we can conclude $\neg\text{K}(\text{DO}(\{a\}_1^{n}, s), \text{DO}(\{a\}_1^{n}, s_0))$. Let $s' = \text{DO}(\{a\}_1^{n-1}, s)$ and $s_{n-1} = \text{DO}(\{a\}_1^{n-1}, s_0)$. So $\text{DO}(\{a\}_1^{n}, s_0) = \text{DO}(a_n, s_{n-1})$. Applying the successor state axiom for K, $\neg\text{K}(\text{DO}(a_n, s'), \text{DO}(a_n, s_{n-1})) \wedge \text{K}(s', s_{n-1}) \Rightarrow \exists\psi(\kappa_\psi^v(a, s_{n-1}) \wedge \neg(\psi(s_{n-1} \Leftrightarrow \psi(s'))))$. The successor state axiom dictates that the only information gained is relative to situation s_{n-1}. Since information was gained about φ, by the same argument used in part 2(b) of the proof for Theorem 3.6 (SSA for KNOW), we can conclude that $\kappa_\varphi^v(a, s_{n-1}) \wedge \varphi(s_{n-1})$. So it is known in situation s_n that $\varphi(s_{n-1})$. But by assumption, the agent knows that $\varphi(s_0)$, so the agent knows that the truth values of $\varphi(s_{n-1})$ and $\varphi(s_0)$ are the same, but prior to situation s_0, the value of $\varphi(s_0)$ was unknown. We can establish that the agent knows that the truth value of φ did not change between situations s_0 and s_{n-1}. Assume otherwise. Because no actions were executed that had unknown causal preconditions, the only ways that φ could have changed would be `T` (known true), `F` (known false), or `U` (unknown and uncorrelated with any other knowledge). In any of these cases, there would be no causal relationship between the old value of φ and the new value, and thus no way to infer its value in s_0 based on its value in s_{n-1}. Since φ didn't change, by definition, **hands-off**(φ). Therefore, $\mathsf{R}_{\{a\}_1^{n}}(\textbf{initially}(\varphi))$.

4. $\mathrm{ACHV}(\varphi, s_0, \{a\}_1^n) \Rightarrow [\varphi(s) \Rightarrow \Sigma^a_\varphi(s) \vee \varphi(s) \wedge \Pi^a_\varphi(s)] \Rightarrow \mathsf{R}_{\{a\}_1^n}(\varphi)$ (By the successor state axiom for φ, and the definitions of $\mathsf{R}_{\{a\}_1^n}(\varphi)$ and $\mathrm{ACHV}(\varphi, s_0, \{a\}_1^n)$.)

A.2.8 Proof of Theorem 3.9 (Updates generated by **cause**$(\varphi,\ \mathtt{T})$*)*

If the only effect of action a executed in state s is of the form **cause**$(\varphi, \mathtt{T})$, then for all P, $\neg\gamma^{\mathtt{F}}_P(a,s)$, $\neg\gamma^{\mathtt{U}}_P(a,s)$ and $\neg\kappa^v_P(a,s)$. Furthermore, for all $P \neq \varphi$, $\neg\gamma^{\mathtt{T}}_P(a,s)$. And finally, $\gamma^{\mathtt{T}}_\varphi(a,s)$. $\mathrm{ACHV}(\pi_a, s, \{\}) \Rightarrow \mathrm{KNOW}(\varphi, \mathrm{DO}(a,s))$. From the above, it follows by definition that $\Sigma^a_\varphi(s)$, $\neg\Sigma^a_{\neg\varphi}(s)$ and $\neg\Pi^a_\varphi(s)$. Furthermore, for all $P \neq \varphi$, $\Pi^a_\varphi(s)$ and $\Pi^a_{\neg\varphi}(s)$.

Since the agent knows the effects of the actions it executes, it is also easy to see that $\mathrm{KNOW}(\Sigma^a_\varphi(s))$ and for all $P \neq \varphi$, $\mathrm{KNOW}(\Pi^a_P(s))$ and $\mathrm{KNOW}(\Pi^a_{\neg P}(s))$. It follows by definition that $\Pi^a_{\mathrm{KNOW}(P)}(s)$ and $\Pi^a_{\mathrm{KNOW}(\neg P)}(s)$.

By the successor state axiom, it follows that $\mathrm{ACHV}(\pi_a, s, \{\}) \Rightarrow \mathrm{KNOW}(\varphi, \mathrm{DO}(a,s))$, but for all $P \neq \varphi$, $\mathrm{ACHV}(\pi_a, s, \{\})(\Rightarrow (\mathrm{KNOW}(P, \mathrm{DO}(a,s)) \Leftrightarrow \mathrm{KNOW}(P,s)) \wedge (\mathrm{KNOW}(P, \mathrm{DO}(a,s)) \Leftrightarrow \mathrm{KNOW}(P,s)))$,

Now, in situation s, φ is either known true, known false or unknown. Collecting the above statements together, we find that in situation $s' = \mathrm{DO}(a,s)$. $\forall P$,

$$P = \varphi \Rightarrow \qquad (\mathrm{KNOW}(P,s) \Rightarrow \mathrm{KNOW}(P,s')) \wedge \tag{A.19}$$

$$(\mathrm{KNOW}(\neg P,s) \Rightarrow \mathrm{KNOW}(\neg P,s')) \tag{A.20}$$

$$(\neg\mathrm{KNOW}(\neg P,s) \wedge \neg\mathrm{KNOW}(P,s)) \Rightarrow \mathrm{KNOW}(P,s') \wedge \tag{A.21}$$

$$P \neq \varphi \Rightarrow \qquad ((\mathrm{KNOW}(P,s') \Leftrightarrow \mathrm{KNOW}(P,s))) \wedge \tag{A.22}$$

$$((\mathrm{KNOW}(\neg P,s') \Leftrightarrow \mathrm{KNOW}(\neg P,s))) \wedge \tag{A.23}$$

If $\mathcal{S} \equiv \{\phi \mid \mathrm{KNOW}(\phi,s)\}$ and $\mathcal{S}' \equiv \{\phi \mid \mathrm{KNOW}(\phi,s')\}$, then we can rewrite the above as $\forall P$,

$$P = \varphi \Rightarrow \quad (P \in \mathcal{S} \vee \neg P \in \mathcal{S} \vee (P \notin \mathcal{S} \wedge \neg P \notin \mathcal{S})) \Rightarrow P \in \mathcal{S}' \tag{A.24}$$

$$P \neq \varphi \Rightarrow \qquad (P \in \mathcal{S}' \Leftrightarrow P \in \mathcal{S})) \wedge \tag{A.25}$$

$$((\neg P \in \mathcal{S}') \Leftrightarrow (\neg P \in \mathcal{S}))\wedge \tag{A.26}$$

Applying the definitions of of the sets $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{U}$, we can rewrite it again as

$$\mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) \cup \mathcal{U}(\varphi, \mathcal{S}) \subseteq \mathcal{T}(\varphi, \mathcal{S}') \tag{A.27}$$

$$\mathcal{S}' \ominus (\mathcal{U}(\varphi, \mathcal{S}) \cup \mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S})) = \mathcal{S} \ominus (\mathcal{U}(\varphi, \mathcal{S}) \cup \mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S})) \tag{A.28}$$

Note that we can replace $\subseteq$ with $=$, since by definition, $\mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) \cup \mathcal{U}(\varphi, \mathcal{S})$ is the set of all literals matching φ. Furthermore, we know that the set of true literals remains unchanged. So we can rewrite this expression as:

$$\mathcal{T}(\varphi, \mathcal{S}') = \mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) \cup \mathcal{U}(\varphi, \mathcal{S}) \tag{A.29}$$

$$\mathcal{S}' \ominus \mathcal{U}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{U}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) \tag{A.30}$$

which is equivalent to $\Delta(\varphi, \mathtt{U} \vee \mathtt{F} \rightarrow \mathtt{T})$

We can easily decompose this into a combination of Domain Growth and Information Gain. Any specific instance of φ in situation s will either be known true, known false, or unknown, so we can divide the above into two cases. In the first case, φ is initially known true or false, so $\mathcal{U}(\varphi,\, \mathcal{S})$ is empty, and we can rewrite the above as

$$\mathcal{T}(\varphi, \mathcal{S}') = \mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{F}(\varphi, \mathcal{S}) \tag{A.31}$$

$$\mathcal{S}' \ominus \mathcal{F}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{F}(\varphi, \mathcal{S}) \tag{A.32}$$

which is the definition of $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{T})$.

In the second case, φ is initially unknown, so $\mathcal{F}(\varphi,\, \mathcal{S})$ is empty:

$$\mathcal{T}(\varphi, \mathcal{S}') = \mathcal{T}(\varphi, \mathcal{S}) \cup \mathcal{U}(\varphi, \mathcal{S}) \tag{A.33}$$

$$\mathcal{S}' \ominus \mathcal{U}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{U}(\varphi, \mathcal{S}) \tag{A.34}$$

Because all literals in $\mathcal{U}(\varphi, \mathcal{S})$ are in $\mathcal{T}(\varphi, \mathcal{S}')$ and no other literals changed membership, $\mathcal{U}(\varphi, \mathcal{S}')$ is empty. Furthermore, since $\mathcal{F}(\varphi, \mathcal{S})$ is empty by assumption, it follows that $\mathcal{F}(\varphi, \mathcal{S}') \supseteq \mathcal{F}(\varphi, \mathcal{S})$. Thus the above set of equations satisfy the definition of $\Delta(\varphi, \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$.

A.2.9 Proof of Theorem 3.10 (Updates generated by **cause**$(\varphi, \mathtt{F})$*)*

The proof is identical to the proof for Theorem 3.9, but with all occurrences of T and F reversed.

A.2.10 Proof of Theorem 3.11 (Updates generated by **cause**$(\varphi, \mathtt{U})$*)*

Suppose the only effect is of the form **cause**$(\varphi, \mathtt{U})$ (*i.e.*, $\gamma^{\mathtt{U}}_{\varphi}(a, s)$). By definition, $\Sigma^{a}_{\varphi}(s) \Leftrightarrow Unk_{\varphi}(a, s)$ and $\Sigma^{a}_{\neg\varphi}(s) \Leftrightarrow \neg Unk_{\varphi}(a, s)$. As in the proof for Theorem 3.9, for all $P \neq \varphi$, $\Pi^{a}_{\varphi}(s)$ and $\Pi^{a}_{\neg\varphi}(s)$.

By the successor state axiom, it follows that
$\mathrm{ACHV}(\pi_a, s, \{\}) \Rightarrow (P(\mathrm{DO}(a, s)) \Leftrightarrow Unk_{\varphi})$. Since $\neg\mathrm{KNOW}(Unk_{\varphi})$, it follows by definition of KNOW that
$\mathrm{ACHV}(\pi_a, s, \{\}) \Rightarrow (\neg\mathrm{KNOW}(P(\mathrm{DO}(a, s))) \wedge \neg\mathrm{KNOW}(\neg P(\mathrm{DO}(a, s))))$. However, for all $P \neq \varphi$, $\mathrm{ACHV}(\pi_a, s, \{\}) \Rightarrow (\mathrm{KNOW}(P, \mathrm{DO}(a, s)) \Leftrightarrow$
$\mathrm{KNOW}(P, s)) \wedge (\mathrm{KNOW}(P, \mathrm{DO}(a, s)) \Leftrightarrow \mathrm{KNOW}(P, s))$,

Collecting the above statements together, we find that in situation $s' = \mathrm{DO}(a, s)$. $\forall P$,

$$P = \varphi \Rightarrow \quad (\neg\mathrm{KNOW}(P, s') \wedge \neg\mathrm{KNOW}(\neg P, s')) \wedge \tag{A.35}$$

$$\begin{aligned} P \neq \varphi \Rightarrow \quad & ((\mathrm{KNOW}(P, s') \Leftrightarrow \mathrm{KNOW}(P, s))) \wedge \\ & ((\mathrm{KNOW}(\neg P, s') \Leftrightarrow \mathrm{KNOW}(\neg P, s))) \wedge \end{aligned} \tag{A.36}$$

Applying the definitions of of the sets $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{U}$, we can rewrite it again as

$$\mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S}) \cup \mathcal{U}(\varphi,\mathcal{S}) \subseteq \mathcal{U}(\varphi,\mathcal{S}') \quad \text{(A.37)}$$

$$\mathcal{S}'\ominus(\mathcal{U}(\varphi,\mathcal{S}) \cup \mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S})) = \mathcal{S}\ominus(\mathcal{U}(\varphi,\mathcal{S}) \cup \mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S})) \quad \text{(A.38)}$$

Note that we can replace $\subseteq$ with $=$, since by definition, $\mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S}) \cup \mathcal{U}(\varphi,\mathcal{S})$ is the set of all literals matching φ. Furthermore, we know that the set of unknown literals remains unchanged. So we can rewrite this expression as:

$$\mathcal{U}(\varphi,\mathcal{S}') = \mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S}) \cup \mathcal{U}(\varphi,\mathcal{S}) \quad \text{(A.39)}$$

$$\mathcal{S}'\ominus\mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S}) = \mathcal{S}\ominus\mathcal{T}(\varphi,\mathcal{S}) \cup \mathcal{F}(\varphi,\mathcal{S}) \quad \text{(A.40)}$$

which is equivalent to $\Delta(\varphi, \mathtt{T} \vee \mathtt{F} \rightarrow \mathtt{U})$

A.2.11 Proof of Theorem 3.12 (Updates generated by **observe***(φ, tv)*

If the only effect of action a executed in state s is of the form **observe**(φ, tv), then for all P, $\neg\gamma^{\mathtt{F}}_P(a,s)$, $\neg\gamma^{\mathtt{U}}_P(a,s)$ and $\neg\gamma^{\mathtt{T}}_P(a,s)$. For all $P \neq \varphi$, $\neg\kappa^v_P(a,s)$, and finally, $\kappa^v_\varphi(a,s)$. From the above, it follows by definition that for all P, $\neg\Sigma^a_P(s)$ and Π^a_P(s). Since the agent knows the effects of the actions it executes, it is also easy to see that $\text{KNOW}(\neg\Sigma^a_P(s))$ and $\text{KNOW}(\Pi^a_P(s))$. It follows by definition that $\Pi^a_{\text{KNOW}(P)}(s)$. Because $\kappa^v_\varphi(a,s)$, and because either $\varphi(s)$ or $\neg\varphi(s)$, it follows by definition that $\Sigma^a_{\text{KNOW}(\varphi)}(s)$. However, since for all $P \neq \varphi$, $\neg\kappa^{tv}_\varphi(a,s)$, it follows that for all $P \neq \varphi$, $\neg\Pi^a_P(s)$. By the successor state axiom, it follows that $\text{ACHV}(\pi_a, s, \{\}) \Rightarrow \text{KNOW}(\varphi, \text{DO}(a,s)) \vee \text{KNOW}(\neg\varphi, \text{DO}(a,s))$. It also follows, for all $P \neq \varphi$, that $\text{ACHV}(\pi_a, s, \{\}) \Rightarrow (\text{KNOW}(\varphi, \text{DO}(a,s)) \Leftrightarrow \text{KNOW}(\varphi, s))$, *i.e.*, there is no change to the agent's knowledge about P. Furthermore, for all P, $\text{ACHV}(\pi_a, s, \{\}) \Rightarrow (\varphi(\text{DO}(a,s)) \Leftrightarrow \varphi(s))$, *i.e.*, there is no change to the world. Since (1) there is no change to the world, (2) the agent's newly gained knowledge is based on the state of the world in situation s and (3) the agent's knowledge in

situation s is assumed to be correct, the successor state axioms further dictate that that the agent's knowledge in situation $\mathrm{DO}(a, s)$ is consistent with its knowledge in situation s: $\mathrm{KNOW}(\varphi, s) \Rightarrow \mathrm{KNOW}(\varphi, \mathrm{DO}(s))$ and
$\mathrm{KNOW}(\neg\varphi, s) \Rightarrow \mathrm{KNOW}(\neg\varphi, \mathrm{DO}(s))$. Collecting the above statements together, we find that in situation $s' = \mathrm{DO}(a, s)$, $\forall P$,

$$P = \varphi \Rightarrow \quad (\mathrm{KNOW}(P, s') \vee \mathrm{KNOW}(\neg P, s')) \wedge \tag{A.41}$$

$$(\mathrm{KNOW}(P, s) \Rightarrow \mathrm{KNOW}(P, s')) \wedge \tag{A.42}$$

$$(\mathrm{KNOW}(\neg P, s) \Rightarrow \mathrm{KNOW}(\neg P, s')) \tag{A.43}$$

$$P \neq \varphi \Rightarrow \quad (\mathrm{KNOW}(P, s') \Leftrightarrow \mathrm{KNOW}(P, s)) \wedge \tag{A.44}$$

$$(\mathrm{KNOW}(\neg P, s') \Leftrightarrow \mathrm{KNOW}(\neg P, s)) \wedge \tag{A.45}$$

If

$$\mathcal{S} \equiv \{\varphi \mid \mathrm{KNOW}(\varphi, s)\}$$

and

$$\mathcal{S}' \equiv \{\varphi \mid \mathrm{KNOW}(\varphi, s')\},$$

then we can rewrite the above as $\forall P$,

$$P = \varphi \Rightarrow \quad P \in \mathcal{S}' \vee \neg P \in \mathcal{S}') \wedge \tag{A.46}$$

$$(P \in \mathcal{S} \Rightarrow P \in \mathcal{S}') \wedge \tag{A.47}$$

$$(\neg P \in \mathcal{S} \Rightarrow \neg P \in \mathcal{S}') \tag{A.48}$$

$$P \neq \varphi \Rightarrow \quad (P \in \mathcal{S}' \Leftrightarrow P \in \mathcal{S})) \wedge \tag{A.49}$$

$$((\neg P \in \mathcal{S}') \Leftrightarrow (\neg P \in \mathcal{S})) \wedge \tag{A.50}$$

Applying the definitions of of the sets $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{U}$, we can rewrite it again as

$$\mathcal{U}(\varphi, \mathcal{S}') = \{\} \wedge \tag{A.51}$$

$$\mathcal{T}(\varphi, \mathcal{S}) \subseteq \mathcal{T}(\varphi, \mathcal{S}') \tag{A.52}$$

$$\mathcal{F}(\varphi, \mathcal{S}) \subseteq \mathcal{F}(\varphi, \mathcal{S}') \tag{A.53}$$

$$\mathcal{S}' \ominus \mathcal{U}(\varphi, \mathcal{S}) = \mathcal{S} \ominus \mathcal{U}(\varphi, \mathcal{S}) \tag{A.54}$$

which is the definition of $\Delta(\varphi, \mathtt{U} \rightarrow (\mathtt{T} \vee \mathtt{F}))$.

A.2.12 Proof of Theorem 3.13 (Information Gain Rule)

First we prove that for any formula, Φ, and literal, φ, if $\textsc{lcw}(\Phi)$ holds before action A is executed and the sole effect of A is $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$, then $\textsc{lcw}(\Phi)$ still holds. Suppose $\textsc{lcw}(\Phi)$ holds and let θ be an arbitrary substitution. By Equation 2.1, we know that $[\mathcal{S} \models \Phi\theta] \vee [\mathcal{S} \models \neg\Phi\theta]$. Since, by the definition of $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$, $\mathcal{S}\,'$ $\ominus\mathcal{U}(\varphi,\, \mathcal{S}) = \mathcal{S}\, \ominus\mathcal{U}(\varphi,\, \mathcal{S})$, and by definition of $\mathcal{U}$, $\mathcal{S}\, \ominus\mathcal{U}(\varphi,\, \mathcal{S}) = \mathcal{S}$, $\mathcal{S} \subseteq \mathcal{S}\,'$. As a result, for any formula Ψ if $\mathcal{S} \models \Psi$ then $\mathcal{S}\,' \models \Psi$. Thus, clearly $[\mathcal{S}\,' \models \Phi\theta] \vee [\mathcal{S}\,' \models \neg\Phi\theta]$. Next we prove that if sole effect of A is $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$, then $\textsc{lcw}(\varphi)$ holds. By the definition of $\Delta(\varphi, \mathtt{U} \rightarrow \mathtt{T} \vee \mathtt{F})$, $\mathcal{U}(\varphi, \mathcal{S}\,') = \{\}$. By the definition of $\mathcal{U}$, $\nexists\theta(\mathcal{S} \not\models \varphi\theta \wedge \mathcal{S} \not\models \neg\varphi\theta)$, or equivalently, $\forall\theta(\mathcal{S} \models \varphi\theta \vee \mathcal{S} \models \neg\varphi\theta)$. But that is exactly the definition of $\textsc{lcw}(\varphi)$.

A.2.13 Proof of Theorem 3.14 (Counting Rule)

Let φ be a literal and suppose that $\mathtt{Cardinality}(\varphi, \mathcal{M}) = \mathtt{Cardinality}(\varphi, \mathcal{W})$. We need show that $\textsc{lcw}(\varphi)$; in other words, we need show that for an arbitrary substitution θ, $[\mathcal{S} \models \varphi\theta] \vee [\mathcal{S} \models \neg\varphi\theta]$. Let M denote the $\{\phi \in \mathcal{M}$ such that ϕ is ground and $\exists\sigma\ \phi = \varphi\sigma\}$. If $\varphi\theta \in \mathcal{M}$ then $\mathcal{S} \models \varphi\theta$ and the proof is complete, so assume that $\varphi\theta \notin \mathcal{M}$. In other words, $\varphi\theta$ is not in M. Let W denote the set $\{\phi \in \mathcal{W}$ such that ϕ is ground and $\exists\sigma\ \phi = \varphi\sigma\}$. Since we assume correct information, $M \subseteq M$, and so by our assumption of cardinality $M = W$. So $\varphi\theta \notin \mathcal{W}$. So $\mathcal{S} \models \neg\varphi\theta$. We conclude $\textsc{lcw}(\varphi)$.

A.2.14 Proof of Theorem 3.15 (Information Loss Rule)

Let Φ be a conjunction of positive literals and suppose that $\textsc{lcw}(\Phi)$. Let φ be a positive literal and let A be an action whose execution leads solely to an update of

the form $\Delta(\varphi, \mathtt{T} \vee \mathtt{F} \rightarrow \mathtt{U})$. To prove that the Information Loss Rule is sound in this case, we need to show that if $\mathrm{LCW}(\Phi)$ no longer holds after executing A then $\Phi \in \mathtt{REL}(\varphi)$ (the set of beliefs removed from $\mathcal{L}$), hence the update correctly recognizes that LCW has been lost, and $\mathcal{L}$ remains conservative. Suppose that $\mathrm{LCW}(\Phi)$ *doesn't* hold after executing A; then there exists a substitution, θ such that $[\mathcal{S}' \not\models \Phi\theta] \wedge [\mathcal{S}' \not\models \neg\Phi\theta]$ even though $[\mathcal{S} \models \Phi\theta] \vee [\mathcal{S} \models \neg\Phi\theta]$. Note that since Φ is conjunctive, $\Phi = \phi_1 \wedge \ldots \wedge \phi_n$. There are two cases:

1. $(\mathcal{S} \models \Phi\theta)$. So forall $\phi_i \in \Phi$ we know that $\mathcal{S} \models \phi_i\theta$. But since $\mathcal{S}' \not\models \Phi\theta$ there exists ϕ_j such that $\mathcal{S}' \not\models \phi_j\theta$. Hence execution of A caused $\Delta(\phi_j\theta, \mathtt{T} \rightarrow \mathtt{U})$. But by the definition of Δ, the only updates produced by A were of the specific form, $\varphi\alpha = \phi_j\theta$. We conclude that $\Phi \in \mathtt{PREL}(\varphi)$.

2. $(\mathcal{S} \models \neg\Phi\theta)$. In this case we know that $\exists\phi_j \in \Phi$ such that $\mathcal{S} \models \neg\phi_j\theta$ yet $\mathcal{S}' \not\models \neg\phi_j\theta$. As above, the restriction on Δ allows us to conclude that $\varphi\alpha = \phi_j\theta =$ and $\Phi \in \mathtt{PREL}(\varphi)$.

To show $\Phi \in \mathtt{REL}(\varphi)$, we now need argue that $\mathcal{M} \cup \mathcal{L} \not\models \neg(\Phi - \phi_j)\theta$. Suppose that this is *not* the case. Since $\mathcal{M}$ and $\mathcal{L}$ are conservative, $\mathcal{S} \models \neg(\Phi - \phi_j)\theta$ as well. Furthermore, since the only change affected by action A had Δ restricted to $\phi_j\theta$, we know that $\mathcal{S}' \models \neg(\Phi - \phi_j)\theta$. But since the falsity of a single conjunct entails the falsity of the whole conjunction (and $\Phi\theta = \phi_j\theta \wedge (\Phi - \phi_j)\theta)$, we conclude that $\mathcal{S}' \models \neg\Phi\theta$. But this contradicts our assumption that A destroyed $\mathrm{LCW}(\Phi)$. So it must be the case that $\mathcal{M} \cup \mathcal{L} \not\models \neg(\Phi - \phi_j)\theta$. Thus $\Phi \in \mathtt{REL}(\varphi)$.

A.2.15 Proof of Theorem 3.16 (Domain Growth Rule)

Let Φ be a conjunction of positive literals and suppose that $\mathrm{LCW}(\Phi)$. Let φ be a positive literal and suppose A is an atomic action whose only effect is $\Delta(\varphi, \mathtt{F} \rightarrow \mathtt{T})$. Suppose that $\mathrm{LCW}(\Phi)$ no longer holds after executing A; then there exists a substitution, θ such that $[\mathcal{S}' \not\models \Phi\theta] \wedge [\mathcal{S}' \not\models \neg\Phi\theta]$ even though

$[\mathcal{S} \models \Phi\theta] \vee [\mathcal{S} \models \neg\Phi\theta]$. A case analysis on the these disjuncts (as in the proof of Theorem 3.15) yields that $\exists\phi_j \in \Phi$ such that $\phi_j\theta = \varphi\alpha$ and that $\Phi \in \texttt{PREL}(\varphi)$. The contradiction argument from that proof also extends to show that $\Phi \in \texttt{REL}(\varphi)$. Now note that after execution of A, we have $\textsc{lcw}(\phi_j\theta)$ (since we know that φ changed to $\texttt{T}$), but by assumption not $\textsc{lcw}(\Phi\theta)$. Therefore, by the contrapositive of Theorem 2.4 (Conjunction), $\neg\textsc{lcw}((\Phi - \phi_j)\theta)$. This leads to $\Phi \in \texttt{MREL}(\varphi)$.

A.2.16 Proof of Theorem 3.17 (Domain Contraction Rule)

Let φ be a positive literal and suppose A is an action whose only effect is $\Delta(\varphi, \texttt{T} \rightarrow \texttt{F})$. To show that the update rule is sound, it is sufficient to prove that for any conjunction of positive literals, $\Phi = \phi_1 \wedge \ldots \wedge \phi_n$, if $\textsc{lcw}(\Phi)$ holds before executing A then $\textsc{lcw}(\Phi)$ holds after executing A. If $\textsc{lcw}(\Phi)$ holds before execution then, for arbitrary θ, we know that $[\mathcal{S} \models \Phi\theta] \vee [\mathcal{S} \models \neg\Phi\theta]$. We need to show that after executing A $[\mathcal{S}' \models \Phi\theta] \vee [\mathcal{S}' \models \neg\Phi\theta]$. Suppose, on the other hand, that $[\mathcal{S}' \not\models \Phi\theta] \wedge [\mathcal{S}' \not\models \neg\Phi\theta]$. But since the Δ effected by A only made *more* atomic formulas false, $\mathcal{S}' \not\models \neg\Phi\theta$ implies $\mathcal{S} \not\models \neg\Phi\theta$. Since $\textsc{lcw}(\Phi)$ holds before executing A, it follows that $\mathcal{S} \models \Phi\theta$ which means that $\mathcal{S} \models \phi_i\theta$ for all $\phi_i \in \Phi$. Now consider the literal φ that has become false.

1. If $\varphi \notin \Phi$ then $\mathcal{S}' \models \Phi\theta$ (since the truth will be unchanged).

2. If $\varphi \in \Phi$ then $\mathcal{S}' \models \neg\Phi\theta$.

Either way there is a contradiction.

A.2.17 Proof of Theorem 3.18 (Tractability of Updates)

The proof was sketched to such an extent in Section 3.5.7 that we will not repeat all details here. Note however, that the exponent $c - 1$ (maximum number of conjuncts in the longest element of $\mathcal{L}$) is the correct one for the following reason. Theorem 2.9 shows that a call to `QueryLCW` with an argument of b conjuncts requires

$O(|\mathcal{L}|^b|\mathcal{M}|^b)$ time. When computing $\texttt{REL}(\varphi)$ or $\texttt{MREL}(\varphi)$ however, the longest argument to `QueryLCW` has $c-1$ conjuncts since the conjunct "x" is removed before the call to `QueryLCW`. (Refer to the definition of `REL` and `MREL`).

A.2.18 Proof of Theorem 3.19 (Minimal Information Loss)

Let φ be a positive literal and let A be an atomic change whose only effect is $\Delta(\varphi, \texttt{T} \vee \texttt{F} \rightarrow \texttt{U})$. Suppose $\Phi \in \texttt{REL}(\varphi)$. We need to show that $\textsc{lcw}(\Phi)$ does not hold after A has occurred. Thus it suffices to show that there exists a θ such that $\mathcal{S}' \not\models \Phi\theta$ and $\mathcal{S}' \not\models \neg\Phi\theta$. Since Φ is conjunctive, the definition of $\texttt{PREL}(\varphi)$ dictates that there exists $\phi \in \Phi$ such that $\phi\theta = \varphi\alpha$. Since the only change from s to s' is that all instances of φ changed their value to unknown, and since from the definition of $\texttt{REL}(\varphi)$, we also have $\mathcal{L} \wedge \mathcal{M} \not\models \neg(\Phi - \phi)\theta$, i.e. all other conjuncts may be true in s, it follows that $\Phi\theta$ may be true in s'. Let $\mathcal{M}'$ denote the state of $\mathcal{M}$ after the update due to A, and let $\mathcal{S}'$ denote the possible states of the world after the update due to A. Since $\Phi\theta$ may be true in s', we have that $\mathcal{S}' \not\models \neg\Phi\theta$. Furthermore, since $\mathcal{M}' \not\models \varphi\alpha$, $\mathcal{M}' \not\models \Phi\theta$, and thus $\mathcal{S}' \not\models \Phi\theta$. Therefore, $\textsc{lcw}(\Phi)$ does not hold.

A.2.19 Proof of Theorem 3.20 (Minimal Domain Growth)

Let φ be a positive literal and let A be an atomic change whose only effect is $\Delta(\varphi, \texttt{F} \rightarrow \texttt{T})$. We need show that if $\Phi \in \texttt{MREL}(\varphi)$ then $\textsc{lcw}(\Phi)$ does not hold after A has occurred. Since Φ is conjunctive, the definition of $\texttt{PREL}(\varphi)$ dictates that there exists $\phi \in \Phi$ such that $\phi\theta = \varphi\alpha$. Since $\Phi \in \texttt{REL}(\varphi)$, we know that $\Phi\theta$ *may* be true in s'. So, $\mathcal{S}' \not\models \neg\Phi\theta$. Since $\Phi \in \texttt{MREL}(\varphi)$, we conclude that $\neg\textsc{lcw}((\Phi - \phi)\theta)$, meaning that for some $\psi \in \Phi$, $\psi\theta \notin \mathcal{M}'$. Hence $\mathcal{M}' \not\models \Phi\theta$, and since Φ contains only positive literals we can conclude that $\mathcal{S}' \not\models \Phi\theta$. Therefore, $\textsc{lcw}(\Phi)$ does not hold.

A.3 Proofs from Chapter 6

A.3.1 Proof of Corollary 6.2 (Causality theorem for **satisfy***)*

First, we show that the corollary follows from the causality theorem. Clause 1 of Pednault's theorem states that there is some point in the plan at which the **satisfy** goal becomes true, before which it is false, and after which it remains true:
$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}} \mathsf{R}_{a_m}(\mathbf{satisfy}(\varphi)) \wedge \alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\mathbf{satisfy}(\varphi))$.
By regressing **satisfy**(φ), we obtain:
$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\Sigma^a_{\mathrm{KNOW}(\varphi)} \vee (\mathbf{satisfy}(\varphi) \wedge \Pi^a_{\mathrm{KNOW}(\varphi)}))\wedge$
$\alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\mathbf{satisfy}(\varphi))$, so
$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\Sigma^a_{\mathrm{KNOW}(\varphi)})$, giving us clause 1(a) of the corollary.
Since the **satisfy** goal is true from s_m to s_n, we can regress $\mathsf{R}_{\{a\}_1^n}(\mathbf{satisfy}(\varphi))$, giving us
$\mathsf{R}_{\{a\}_1^{n-1}}(\Sigma^a_{\mathrm{KNOW}(\varphi)} \vee (\mathbf{satisfy}(\varphi) \wedge \Pi^a_{\mathrm{KNOW}(\varphi)}))$.
Since $\Sigma^a_{\mathrm{KNOW}(\varphi)} \Rightarrow \Pi^a_{\mathrm{KNOW}(\varphi)}$, this gives us
$\mathsf{R}_{\{a\}_1^{n-1}}(\Pi^a_{\mathrm{KNOW}(\varphi)})$.
By definition (Eqn 3.25), $\Pi^a_{\mathrm{KNOW}(\varphi)} \Rightarrow \mathrm{KNOW}(\Pi^a_{\varphi})$, giving us clause 1(b) of the corollary.
Clause 2 of Pednault's causality theorem states that the goal is true at all time points. Regressing **satisfy**(φ) back to the initial state, we find that $\alpha \models \mathrm{KNOW}(\varphi)$, and for all $i = m \ldots n-1$ $\alpha \models \mathsf{R}_{\{a\}_1^i}(\Pi^{a_{i+1}}_{\mathrm{KNOW}(\varphi)})$, giving us Clause 2 of the corollary.
We now prove the "only if" case by starting from the corollary and showing that the causality theorem follows. If clause 2 of the corollary holds, then
$\alpha \models \mathrm{KNOW}(\varphi)$, and for all $i = 1 \ldots n-1$ $\alpha \models \mathsf{R}_{\{a\}_1^i}(\Pi^{a_{i+1}}_{\mathrm{KNOW}(\varphi)})$.
Applying the regression operator for **satisfy** in the forward direction, we find that $\alpha \models \mathrm{KNOW}(\varphi) \Rightarrow \mathsf{R}_{\{\}}(\mathbf{satisfy}(\varphi))$ and it follows by induction for each prefix $\{a\}_1^i$ that

$\alpha \models \mathsf{R}_a\{a\}_1^{i-1}(\mathbf{satisfy}(\varphi) \wedge \Pi^{a_{i+1}}_{\text{KNOW}(\varphi)}) \Rightarrow \mathsf{R}_a\{a\}_1^{i}(\mathbf{satisfy}(\varphi))$, satisfying clause 2. If clause 2 of Pednaut's theorem *doesn't* hold, $\alpha \not\models \text{KNOW}(\varphi)$, or for some $i < n$, $\alpha \not\models \mathsf{R}_{\{a\}_1^i}(\Pi^{a_{i+1}}_{\text{KNOW}(\varphi)})$, and clause 1 must hold. For some m,

1. $\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\Sigma^{a_m}_{\text{KNOW}(\varphi)})$
2. For all $i = m \ldots n-1$ $\alpha \models \mathsf{R}_{\{a\}_1^i}(\Pi^{a_{i+1}}_{\text{KNOW}(\varphi)})$.

Consider the smallest m for which the above holds. By the regression operator for **satisfy**, $\mathsf{R}_{\{a\}_1^{m-1}}(\Sigma^{a_m}_{\text{KNOW}(\varphi)}) \Rightarrow \mathsf{R}_{\{a\}_1^m}(\mathbf{satisfy}(\varphi))$.
By applying regression forward once more, we find
$\mathsf{R}_{\{a\}_1^m}(\Pi^{a_{m+1}}_{\text{KNOW}(\text{KNOW}(\varphi))} \wedge \mathbf{satisfy}(\varphi)) \Rightarrow \mathsf{R}_{\{a\}_1^{m+1}}(\mathbf{satisfy}(\varphi))$, and by induction, the same holds for all $m+1 \leq i \leq n$. This gives us clause 2(b) of the causality theorem.
We picked the smallest m satisfying clause 1 of the corollary, so we know that for all earlier time points, KNOW(φ) is either not established or not subsequently preserved. It follows from the regression operator for **satisfy** that
$\alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\mathbf{satisfy}(\varphi))$. This gives us clause 1(b) of the causality theorem.

A.3.2 Proof of Lemma 6.3

By definition, **initially**(φ) is true in state s_m iff $\forall s \in \text{ORIG}_m.\ P(s)$. But $\text{ORIG}_m \supseteq \text{ORIG}_{m+i}$, since the agent cannot forget about what it knows of previous situations (this follows directly from Theorem 3.2 (SSA for K). So **initially**(φ) must hold in situation s_{m+i} as well.

A.3.3 Proof of Lemma 6.4

The proof is by induction on n. The base case is trivial. $\mathsf{R}_{\{\}}(\textbf{hands-off}(\varphi))$ is true by definition. We will assume that $\mathsf{R}_{\{a\}_1^{k-1}}(\textbf{hands-off}(\varphi))$ and show that $\mathsf{R}_{\{a\}_1^{k}}(\textbf{hands-off}(\varphi))$.

$\mathsf{R}_{\{a\}_1^{k-1}}(\text{KNOW}(\Pi^{a_k}_{\neg\varphi}) \wedge \text{KNOW}(\Pi^{a_k}_{\varphi}))$ by assumption. Combining this with the induction hypothesis, we get

$\mathsf{R}_{\{a\}_1^{k-1}}(\text{KNOW}(\Pi^{a_k}_{\neg\varphi}) \wedge \text{KNOW}(\Pi^{a_k}_{\varphi}) \wedge \mathsf{R}_{\{a\}_1^{k-1}}(\textbf{hands-off}(\varphi))$ By the definition of regression for **hands-off**, we get $\mathsf{R}_{\{a\}_1^{k-1}}\mathsf{R}_{a_k}(\textbf{hands-off}(\varphi))$, which is equivalent to $\mathsf{R}_{\{a\}_1^{k}}(\textbf{hands-off}(\varphi))$, which is what we wanted to prove.

A.3.4 Proof of Lemma 6.5

By the contrapositive of Lemma 6.3, if $\mathsf{R}_{\{a\}_1^n}(\textbf{initially}\varphi)$ is false, so is $\mathsf{R}_a\{a\}_1^i(\textbf{initially}\varphi)$, for $0 < i < n$. Combining this fact with the regression operator for **hands-off** gives us $\mathsf{R}_{\{a\}_1^{n-1}}(\text{KNOW}(\Pi^{a_{i+i}}_{\neg\varphi}) \wedge \text{KNOW}(\Pi^{a_{i+i}}_{\varphi}) \wedge \textbf{hands-off}(\varphi))$. Continuing to regress on $\textbf{hands-off}(\varphi)$ in this manner yields

$\mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi^{a_{i+1}}_{\neg\varphi}) \wedge \text{KNOW}(\Pi^{a_{i+1}}_{\varphi}))$ for all $0 < i < n$,

A.3.5 Proof of Corollary 6.6 (Causality theorem for ***initially****)*

First we prove that the corollary follows from the causality theorem. Clause 1 of Pednault's theorem states that there is some point in the plan before which the **initially** goal is false, and after which it is true, so

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}\mathsf{R}_{a_m}(\textbf{initially}(\varphi)) \wedge \alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{initially}(\varphi))$

By regressing $\textbf{initially}(\varphi)$, we get:

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{initially}(\varphi) \vee (\kappa^{tv}_{\varphi}(a_m) \wedge \varphi \wedge \textbf{hands-off}(\varphi)))) \wedge \alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{initially}(\varphi))$

Simplifying, we get:

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\kappa^{tv}_{\varphi}(a_m) \wedge \varphi \wedge \textbf{hands-off}(\varphi)))$

Since $\alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{initially}(\varphi))$, we by Lemma 6.5,

$\mathsf{R}_{\{a\}_1^{m-1}}(\textbf{hands-off}(\varphi)) \Leftrightarrow \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi^{a}_{\neg\varphi}) \wedge \text{KNOW}(\Pi^{a}_{\varphi}))$ for all $0 < i < m$.

So we can rewrite the above formula as:

1. $\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\kappa^{tv}_{\varphi}(a_m) \wedge \varphi \wedge$

2. $\alpha \models \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi^a_{\neg\varphi})) \wedge (\text{KNOW}(\Pi^a_{\varphi}))$ for all $0 < i < m$,

which is the same as item 1 of the corollary.

Item 2 of Pednault's causality theorem states that the goal is true in the initial state and remains true thereafter. By Lemma 6.3, it will necessarily remain true, so we only care that it is true in the initial state. By the definition of regression on **initially**(φ), this is the case if and only if KNOW(φ) is true in the initial state, which is equivalent to item 2 of the corollary.

Next we show the reverse case, *i.e.*, that the causality theorem follows from the corollary. If clause 2 of the corollary applies, then $\alpha \models \text{KNOW}(\varphi)$, and by definition, $\alpha \models \mathsf{R}_a\{\}(\textbf{initially}(\varphi))$. By Lemma 6.3, the same holds for all plans of any length, thus satisfying clause 2 of the causality theorem.

Otherwise, $\alpha \not\models \text{KNOW}(\varphi)$, so clause 1 must hold. By clause 1, at some point, the agent must observe φ. Without loss of generality, consider the earliest such time point, *i.e.*, the smallest m for which

1. $\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\kappa^{tv}_{\varphi}(a_m) \wedge \varphi \wedge$

2. $\alpha \models \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi^a_{\neg\varphi})) \wedge (\text{KNOW}(\Pi^a_{\varphi}))$ for all $0 < i < m$,

By Lemma 6.4, $\mathsf{R}_{\{a\}_1^i}(\textbf{hands-off}(\varphi))$, for all $0 < i < m$. By the regression operator for **initially**,

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\kappa^{tv}_{\varphi}(a_m) \wedge \varphi \wedge \textbf{hands-off}(\varphi))) \Rightarrow \alpha \models \mathsf{R}_{\{a\}_1^m}(\textbf{initially}(\varphi))$.

Since we chose the earliest time at which φ was observed, and since $\alpha \not\models \text{KNOW}(\varphi)$, it follows by regression on **initially** that $\alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{initially}(\varphi))$.

Thus clause 1 of the causality theorem is satisfied.

A.3.6 Proof of Corollary 6.7 (Causality theorem for **hands-off***)*

First we prove that the corollary follows from the causality theorem. Clause 1 of Pednault's theorem states that there is some point in the plan at which the

hands-off goal becomes true, before which it is false.

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}} \mathsf{R}_{a_m}(\textbf{hands-off}(\varphi)) \wedge \alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{hands-off}(\varphi))$ However, this contradicts the definition of **hands-off**: By regressing **hands-off**(φ), we obtain:

$\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{hands-off}(\varphi) \wedge \ldots) \wedge \alpha \not\models \mathsf{R}_{\{a\}_1^{m-1}}(\textbf{hands-off}(\varphi))$, a contradiction.

So clause 1 of the causality theorem is always false for **hands-off**. Turning to clause 2, we see that the goal is true in the initial state and remains true thereafter:

$\alpha \models \mathsf{R}_{\{a\}_1^{m}}(\textbf{hands-off}(\varphi))$, for all $i = 1 \ldots n-1$. There are two cases. If neither **initially**(φ) nor **initially**($\neg\varphi$) is achieved at any point, then by Lemma 6.4, for all $i = 1 \ldots n-1$, $\mathsf{R}_{\{a\}_1^{n-1}}(\textbf{hands-off}(\varphi)) \Rightarrow \mathsf{R}_{\{a\}_1^{i}}(\mathrm{KNOW}(\Pi^{a}_{\neg\varphi}) \wedge \mathrm{KNOW}(\Pi^{a}_{\varphi}))$, which corresponds to clause 1 of this theorem.

Suppose that either **initially**(φ) or **initially**($\neg\varphi$) is achieved at some point. Without loss of generality, we will assume that **initially**(φ) is true. The other case is symmetric. If **initially**(φ) is achieved at some point, then we know from Lemma 6.3 that it is true at all time points afterward. Thus we can regress **hands-off** as follows until the **initially** condition is no longer met.

By the definition of regression on **hands-off**, $\alpha \models \mathsf{R}_{\{a\}_1^{n-1}}((\mathrm{KNOW}(\Pi^{a}_{\neg\varphi}) \vee$ **initially**$(\varphi)) \wedge (\mathrm{KNOW}(\Pi^{a}_{\varphi}) \vee$ **initially**$(\neg\varphi)) \wedge$ **hands-off**$(\varphi))$. But since **initially**(φ) is true, this simplifies to $\alpha \models \mathsf{R}_{\{a\}_1^{m-1}}(\mathrm{KNOW}(\Pi^{a}_{\varphi}) \wedge \textbf{hands-off}(\varphi))$. Continuing to regress on **hands-off**, we get $\mathrm{KNOW}(\Pi^{a_{i+1}}_{\varphi})$ for all situations after the **initially** goal was met. Regressing on **hands-off** beyond that point is unnecessary, as a casual inspection of the regression operator for **initially** reveals. That is, **initially** regresses to **hands-off**. Because of the **hands-off**, we know that $\mathrm{KNOW}(\Pi^{a_{i+1}}_{\varphi})$ will in fact hold for all situations before the **initially** goal is achieved. Thus **initially**(φ) is true in the final state, and $\mathrm{KNOW}(\Pi^{a_{i+1}}_{\varphi})$ is true for all situations.

Next we will show that the causality theorem follows from the corollary. First, consider clause 1 of the corollary:

For all $i = 1 \ldots n-1$ $[\alpha \models \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi_{\varphi}^{a_{i+1}}) \wedge \Pi_{\neg\varphi}^{a_{i+1}})]$.
By Lemma 6.5, $\alpha \models \mathsf{R}_{\{a\}_1^i}(\textbf{hands-off}(\varphi))$, for all i.
If clause 2 holds, then either

1. $\alpha \models \mathsf{R}_{\{a\}_1^n}(\textbf{initially}(\varphi))$ and for all $i = 1 \ldots n-1$
   $\alpha \models \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi_{\varphi}^{a_{i+1}}))$ OR

2. $\alpha \models \mathsf{R}_{\{a\}_1^n}(\textbf{initially}(\neg\varphi))$ and for all $i = m \ldots n-1$
   $\alpha \models \mathsf{R}_{\{a\}_1^i}(\text{KNOW}(\Pi_{\neg\varphi}^{a_{i+1}}))$

In either case, by the regression operator for **hands-off**,
$\alpha \models \mathsf{R}_{\{a\}_1^{i+1}}(\textbf{hands-off}(\varphi))$. We can continue regressing **hands-off** all the way back to the initial state, thus giving us clause 2 of the causality theorem.

A.3.7 Proof of Theorem 6.8 (Correctness of HandleGoals)

We show that for every goal given to it, the HandleGoals algorithm (Figure 5.2) replaces the goal with structures that ensure, by the causality theorem, that the goal will be achieved.
If line 1 applies then the goal is of the form $P \Rightarrow Q$, where $P \models Q$, so the goal is true tautologically.
If line 2 applies, then the goal is of the form **hands-off**(φ). HandleGoals achieves this goal by calling AddLink, which adds a causal link to $\mathcal{C}$, protecting both φ and $\neg\varphi$. As is discussed in the proof for Theorem 6.9, this causal link ensures that the conditions specified in the causality theorem for **hands-off** are true.
Otherwise, line 3 must apply. Line 3(a) reduces the goal to one that is logically equivalent. Lines 3(b) and 3(c) correspond to Pednault's Expansion Theorem: 3(a) adds a new step to the plan that will make the goal true. 3(b) ensures that either an existing step or the initial state (represented by the dummy step A_0) supports

the goal. In either case, various structures are added to the plan to satisfy the Causality Theorem.

The proof will proceed by considering each each class of goals in turn, showing that HandleGoals satisfies the Causality Theorem for each.

- **satisfy**(φ): Only line three of HandleGoals applies. By MGU, the only effects that will be selected are those annotated with **cause** or **observe**, matching φ. In the call to AddLink MGU(e,g) will be added to $\mathcal{B}$, ensuring that the effect does indeed match the goal and the preconditions of the effect (along with the preconditions of the step, if it was just added) will be added to $\mathcal{G}$. An ordering constraint is added to $\mathcal{O}$ to ensure that the action producing the effect is executed before the time when the goal is required,

  Recall that preconditions for **cause**(φ) are of the form $\gamma^{\mathtt{T}}_{\varphi}(a)$. Preconditions for **observe**(φ) are of the form $\kappa^{\mathtt{T}}_{P}(a)$. These preconditions are added to $\mathcal{G}$, ensuring that they will either be achieved by the planner, or will be assumed to be true and verified by subsequent observation some time before the **satisfy** goal must be achieved. In either case, the agent will know whether the desired effect occurred. We will first assume that these (assumed) preconditions are verified immediately after the action is executed, conforming to the definition of $\Sigma^{a}_{\mathrm{KNOW}(\varphi)}$. Then we show in the item **assumptions** below that we can relax that assumption, using the causal links to keep track of correlations. We will take it as a given that any assumed conditions actually hold, since if they don't, that fact will be detected and the plan will be rejected.

  If the precondition of **cause** is not assumed, then the agent will know before executing the action that the condition is true: $\mathrm{KNOW}(\gamma^{\mathtt{T}}_{\varphi}(a))$. If an assumed precondition of a **cause** effect is verified by that effect, then we have $\kappa^{tv}_{\gamma^{\mathtt{T}}_{\varphi}(a)}(a) \wedge \gamma^{\mathtt{T}}_{\varphi}(a)$.

One precondition of **observe**(φ) that is always assumed is φ itself. The remaining preconditions may be assumed as well. $\kappa^{tv}_{\varphi}(a) \wedge \varphi$. The condition $\Pi^{s}_{\mathrm{KNOW}(\varphi)}$ is ensured by the causal link. If S_p threatened that condition, then it would violate the causal link that it itself provided.

Putting all this together, we get the definition of $\Sigma^{a}_{\mathrm{KNOW}(\varphi)}$: $\mathrm{KNOW}(\gamma^{\mathtt{T}}_{\varphi}(a)) \vee (\kappa^{tv}_{\gamma^{\mathtt{T}}_{\varphi}(a)}(a) \wedge \gamma^{\mathtt{T}}_{\varphi}(a)) \vee (\kappa^{tv}_{\varphi}(a) \wedge \varphi \wedge \Pi^{a}_{\mathrm{KNOW}(\varphi)})$ Thus satisfying clause 1(a) and the first part of clause 2 of the Causality Theorem for **satisfy**. The remainder of the Causality Theorem is provided by Theorem 6.9.

- **initially**(φ): If φ is known to be true in the initial state, then HandleGoals will not add any additional structures to the plan. This is correct, since condition 2 of the causality theorem for **initially** is satisfied. Otherwise, HandleGoals will select a new or existing action that has an effect unifying with the goal. `MGU` will only choose matching **observe** effects in support of **initially** goals, and will return the binding constraints that must be satisfied for the match to succeed. HandleGoals adds these constraints to $\mathcal{B}$ and adds to $\mathcal{G}$ the preconditions needed to ensure that the effect will occur. As discussed in the case for **satisfy** above, the **observe** effect ensures that $\kappa^{tv}_{\varphi}(a) \wedge \varphi$, thus satisfying condition 1(a) of the causality theorem for **initially**. Unlike **satisfy**, there's no need to ensure that φ is still true after execution, the the causal link extending forward from the effect is not required. However, a causal link is added extending from the initial step to the producer, satisfying condition 1(b) of the Causality Theorem for **initially**.

- $\forall$ goals: If line 3(a) of HandleGoals is chosen, then line 2 of `Reduce` applies. Both reductions replace g with goals that are logically equivalent. Item 2(b) replaces g with the universal base. Since the universal base of a goal is

equivalent to the original goal, and since LCW(Φ) makes it possible to correctly compute the universal base, this results in no logical change to $\mathcal{G}$. Line 2(b) replaces the goal g with two equivalent goals, one of the form $\varpi \Rightarrow g$, and the other $\neg\varpi \Rightarrow g$. The two contexts are mutually exclusive and exhaustive, so the two new goals together are equivalent to g. If line 3(b) or 3(c) of HandleGoals is chosen, then by MGU, the supporting effect must entail the goal. Any restrictions on the effect from the **when** clause will be added as a full universally quantified goal to $\mathcal{G}$. If the precondition is entailed by the context of the goal, it will be removed in Line 1. Otherwise, it must become true for all members of the universe.

- LCW goals: Line 1(a) of Reduce is equivalent to the composition rule, which is proved sound in Theorem 2.4. Line 1(c) is the same as the conjunction rule, a corollary of the composition rule. Lines 3(b) and 3(c) of HandleGoal call MGU, which only succeeds if there is a perfect match, which corresponds to be instantiation rule, proved sound in Theorem 2.2.

- **assumptions**: The successor state axiom for KNOW (Theorem 3.6) requires that if a condition is made true by an action, its preconditions be known — immediately after the action is executed — to have held immediately prior. The assumption mechanism used by PUCCINI violates this condition. The reason for the difference is that Theorem 3.6 is based on the assumption that correlations between unknown fluents are forbidden, and delaying the verification of the preconditions introduces such correlations. However, the causal link mechanism used by the planner provides a limited ability to keep track of such correlations, so delaying the verification is allowed.

  Specifically, there is a correlation between the truth value of the precondition immediately before execution and the truth value of the effect immediately

after. This correlation persists until either the precondition or the postcondition is affected, so if the precondition is later observed, that reveals the truth value of the effect. The causal link between the assumed precondition and the observation used to verify it both records the fact that the correlation exists and ensures that the precondition isn't affected before it's observed. The effect condition is also protected – by the causal link to the precondition that it supports (it must support some condition or the planner wouldn't be trying to make it true). The effect condition is guaranteed to persist at least until the assumed precondition is observed, since the the observation is required to occur before the assumed effect is needed.

A.3.8 Proof of Theorem 6.9 (Correctness of HandleThreats)

The purpose of HandleThreats (Figure 5.6) is to maintain the preservation conditions required by be causality theorem, which are recorded in the plan using causal links. There are two things to prove: that all threats are detected, and that each refinement for resolving threats makes the threat go away. We will prove the latter first, by considering each line of HandleThreats in turn.

1. Promotion: By the causality theorem, the truth of a condition at a given time depends only on what happened before that time, not on what follows. So an action executed at a time point after the goal is required to be true cannot violate the causality theorem for that goal.

2. Demotion: Demotion can never succeed for **initially** or **hands-off**, since the causal link extends all the way to the initial state. By the causality theorem for **satisfy**, moving the threatening action before the point at which the enabling precondition is satisfied eliminates the threat, since the condition only needs to be preserved after it becomes true.

3. Confrontation: Making sure that the $\neg\Sigma^a_{\neg\varphi}$ is equivalent to ensuring that Π^a_{φ}.

4. LCW threats:

   (a) If $P_1(\texttt{foo})$ is true, then $\text{LCW}(P_1(\texttt{foo}))$ is true. If $\text{LCW}(P_2(\texttt{foo})) \wedge \ldots \wedge \text{LCW}(P_n(\texttt{foo}))$ then by the conjunction theorem, $\text{LCW}(P_1(\texttt{foo}) \wedge \ldots (P_n(\texttt{foo}))$. Since all that changed was $P_1(\texttt{foo})$ and $\text{LCW}(P_1(x) \wedge \ldots \wedge P_n(x))$ held before the change, by the instantiation theorem, $\text{LCW}(P_1(x_i) \wedge \ldots \wedge P_n(x_i))$ is true for all $x_i \neq \texttt{foo}$. Since it is also true for $x_i = \texttt{foo}$, it is true for all x_i.

   (b) If any of $P_2(\texttt{foo}), \ldots, P_n(\texttt{foo})$ is false then $\text{LCW}(P_2(\texttt{foo}) \wedge \ldots \wedge P_n(\texttt{foo}))$ is true, and by the argument above, $\text{LCW}(P_1(x) \wedge \ldots \wedge P_n(x))$ is true.

5. An implication can be made true by making the antecedent false or the consequent true, so each of the updates mentioned makes $\forall x[q_1(\texttt{foo}) \Rightarrow S(\texttt{foo})]$ true. Since this formula was originally true for all x, and the only threat came from a change to $P_1(\texttt{foo})$, it is true for all $x_i \neq \texttt{foo}$. Since it is also true for $x_i = \texttt{foo}$, it is true for all x_i.

6. **hands-off**: The handsoff condition is threatened because the **cause** effect violates clause 1 of the causality theorem for **hands-off**. But adding the subgoal **initially**(φ) ensures that clause 2 is satisfied, so the threat goes away.

We now show that all threats are detected by HandleThreats. A condition is threatened in a plan P whenever the preservation conditions in the causality theorem are violated in some refinements of the plan. These threats are detected by examining the causal links added during goal establishment. It is shown in theorem 6.8 that the intervals and conditions recorded by these links correspond to the requirements of the causality theorem. We will show here that the possible ways that the links can be violated are in fact detected.

1. **satisfy**: The only conditions that violate the presevation condition KNOW(Π^a) are of the form **cause**(φ, T) or **cause**(φ, U). Both are detected.

2. **initially** and **hands-off**: The only conditions that violate KNOW(Π^a) are of the form **cause**(φ, T), **cause**(φ, F) or **cause**(φ, U). All three are detected.

3. $\forall$: $\forall$ goals are always implications. The only way to violate an implication is to make the antecedent true or the consequent falls. The latter is detected as an **initially** or **satisfy** threat, whichever is the case. The former is detected in line 5 of HandleForall.

4. LCW: By Theorems 3.13 through 3.17, the only way to invalidate an LCW formula is through Domain Growth or Information Loss. Both of these cases are detected in line 4 of HandleThreats.

*A.3.9 Proof of Lemma 6.11 (*PUCCINI *loop invariant (initial))*

In the initial call to PUCCINI, $\mathcal{G}$ contains the high-level goal γ. So if the conditions on $\mathcal{G}$ are satisfied in the initial call to PUCCINI, and the current plan (consisting of no actions) is executed, then γ is trivially satisfied. Since no actions have been executed, the executed prefix of the plan is trivially consistent with the state of the world.

*A.3.10 Proof of Lemma 6.12 (*PUCCINI *loop invariant)*

We will show that each recursive control path of PUCCINI preserves the loop invariant.

1. HandleGoals, followed by HandleThreats: A goal is removed from the $\mathcal{G}$, but by Theorems 6.8) and 6.9, HandleGoals HandleThreats together ensure that the Causality Theorem is obeyed, and HandleThreats ensures that the Causality

Theorem for all previously removed goals continues to to be obeyed. Thus, the loop invariant is preserved.

2. HandleExecution: IsExecutable? only returns true when the preconditions of the action are true, so executing the action is well-defined. Because some effects may be supported by assumptions, not all effects are guaranteed to be valid, but if any assumption the executed action *supports* (including self-links) is not true, this will be detected due to the observational effects supporting the assumptions, and the plan will be rejected, thus preserving the loop invariant requirement that executing the remainder of the plan will succeed, provided that the assumptions supported by *un*executed actions are valid.

*A.3.11 Proof of Lemma 6.12 (*PUCCINI *loop invariant true on termination)*

PUCCINI will only terminate with success if all flaws have been fixed, in which case it makes no further modifications to the plan. Since the loop invariant was true on the previous iteration, it is true afterward.

A.3.12 Proof of Theorem 6.14 (Soundness of PUCCINI*)*

Lemmas 6.11-6.13 combine to form a simple inductive argument that PUCCINI is sound. Since the loop invariant holds on the initial call to PUCCINI and on each subsequent iteration, it holds when PUCCINI terminates with success. Since PUCCINI only terminates with success if $\mathcal{G}$ is empty and all actions have been executed, it follows from the loop invariant that the goal Γ has in fact been achieved.

Appendix B

SOFTBOT DOMAIN

Following are some of the more important operators, (aka action schemas) from the UNIX domain. The Lisp-style syntax differs from the infix notation used in the body of the thesis, but the correspondence should be obvious. The `interface` block collects all parts of the operator that concern interaction with the outside world. The `exec-func` and `translation` together correspond to the `execute` field in the SADL EBNF (Table 3.2), and the `sense-func` corresponds to the `sense` field in the EBNF.

B.1 File Operators

```
(defoperator CD ((directory ?d) (path ?n))
  (declare ((directory ?old) (path ?oldpath)))
  (documentation "Change the current working directory")
  (precond (satisfy (pathname ?d ?n)))
  (undo-cond (and (satisfy (current.directory ?old))
                  (satisfy (pathname ?old ?oldpath))))
  (interface ((exec-func execute-unix-command)
              (translation ("cd " ?n))
              (undo-trans ("cd " ?oldpath))
              (error-func default-unix-error?)
              (terminate-detect read-unix-prompt)))
  (effect (cause (current.directory ?d))))
```

```
(defoperator FIND-FILE ((path ?pathname) (file !file))
    (precond (satisfy (current.shell csh)))
    (effect (observe (pathname !file ?pathname)))
    (interface ((exec-func execute-unix-command)
                (sense-func ( (!file) (find-file-sense :new)))
                (translation ("ls -dF " ?pathname))
                (sensor-bind-func ucpop::my-get-sense-bindings)
                (error-func no-error))))


(defoperator LS ((directory ?d) (path ?dp))
  (precond (and (satisfy (current.shell csh))
(satisfy (pathname ?d ?dp))))
  (effect (forall ((file !f))
                 (exists ((path !p) (filename !n))
                         (when (parent.directory !f ?d)
                           (and (observe (parent.directory !f ?d))
                                (observe (pathname !f !p))
                                (observe (filename !f !n)))))))
  (interface ((exec-func execute-unix-command)
              (translation ("ls -F " ?dp))
              (sense-func ((!f !n !p) (ls-sense ?dp :new)))
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (error-func default-unix-error?)
              (terminate-detect read-unix-prompt))))
```

```
(defoperator PWD ((directory !dir))
  (declare ((path !path)
            (filename !name)))
  (precond (satisfy (current.shell csh)))
  (effect (when (current.directory !dir)
            (and (observe (current.directory !dir))
                 (observe (pathname !dir !path))
                 (observe (filename !dir !name)))))
  (interface ((exec-func execute-unix-command)
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (sense-func ((!path !name !dir) (pwd-sense :new)))
              (translation ("pwd")))))


(defoperator WC ((simple.file ?x) (number !char) (number !word)
                 (number !line) (directory ?dir))
  (declare ((filename ?name)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (current.directory ?dir))
                (satisfy (parent.directory ?x ?dir))
                (satisfy (filename ?x ?name))))
  (effect (and (observe (character.count ?x !char))
               (observe (word.count ?x !word))
               (observe (line.count ?x !line))))
  (interface ((exec-func execute-unix-command)
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (sense-func ((!char !word !line) (wc-sense)))
```

```
            (error-func default-unix-error?)
            (translation ("wc \"" ?name "\"")))))


(defoperator GREP ((string ?string) (simple.file ?x) (bool !grep-t))
  (declare ((filename ?nm) (directory ?d)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (current.directory ?d))
                (satisfy (filename ?x ?nm))
                (satisfy (parent.directory ?x ?d))))
  (effect (observe (string.in.file ?string ?x) !grep-t))
  (interface ((exec-func execute-unix-command)
              (sense-func ((!grep-t) grep-sense))
              (error-func no-error)
              (translation ("grep " ?string " " ?nm)))))


(defoperator OWNER-OF-FILE ((file ?x) (userid !owner))
  (declare ((filename ?nm) (directory ?d)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (current.directory ?d))
                (satisfy (parent.directory ?x ?d))
                (satisfy (filename ?x ?nm))))
  (effect (and (observe (owner ?x !owner))))
  (interface ((exec-func execute-unix-command)
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (sense-func ((!owner) owner-of-file-sense))
              (translation ("ls -lLd " ?nm )))))
```

```
(defoperator PROTECTION-ON-FILE ((file ?x) (bool !g-read)
                                (bool !g-write) (bool !g-exec))
  (declare ((path ?nm)))
  (precond (and (satisfy (current.shell  csh))
                (satisfy (pathname ?x ?nm))))
  (effect (and (observe (group.protection ?x readable) !g-read)
               (observe (group.protection ?x writeable) !g-write)
               (observe (group.protection ?x executable) !g-exec)))
  (interface ((exec-func execute-unix-command)
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (sense-func ((!g-read !g-write !g-exec)
                   (protect-on-file-sense)))
              (translation ("ls -lLd " ?nm )))))


(defoperator GROUP-PROTECT-FILE ((file ?x))
  (declare ((path ?nm)
            (userid ?person)))
  (precond (and (satisfy (current.shell  csh))
                (satisfy (current.userid  ?person))
                (satisfy (owner ?x ?person))
                (satisfy (pathname ?x ?nm))))
  (effect (and (cause (group.protection ?x readable))
                (cause (group.protection ?x writeable))
                (cause (group.protection ?x executable))))
  (interface ((exec-func execute-unix-command)
```

```
              (translation ("chmod g+rwx " ?nm )))))


(defoperator GROUP-UNPROTECT-FILE ((file ?x))
  (declare ((path ?nm)
            (userid ?person)))
  (precond (and (satisfy (current.shell  csh))
                (satisfy (current.userid  ?person))
                (satisfy (owner ?x ?person))
                (satisfy (pathname ?x ?nm))))
  (effect (and (cause (group.protection ?x readable) F)
                (cause (group.protection ?x writeable) F)
                (cause (group.protection ?x executable) F)))
  (interface ((exec-func execute-unix-command)
              (translation ("chmod g-rwx " ?nm )))))


(defoperator MOVE-DIR ((file ?x) (directory ?d2))
  (declare ((filename ?nm)
            (path ?d2path)
            (directory ?d)))
  (precond (and (satisfy (current.directory  ?d))
                (satisfy (pathname ?d2 ?d2path))
                (satisfy (parent.directory ?x ?d))
                (satisfy (filename ?x ?nm))
                (satisfy (current.shell  csh))))
  (effect (cause (parent.directory ?x ?d2)))
  (interface ((exec-func execute-unix-command)
```

```
              (translation ("mv " ?nm " " ?d2path)))))


(defoperator COMPARE-LENGTH ((file ?f1) (file ?f2) (number ?l1)
                             (number ?l2) (bool !t))
  (declare (directory ?d))
  (precond (and (satisfy (parent.directory ?f1 ?d))
                (satisfy (parent.directory ?f2 ?d))
                (satisfy (character.count ?f1 ?l1))
                (satisfy (character.count ?f2 ?l2))))
  (effect (observe (same.length ?f1 ?f2) !t))
  (interface ((exec-func execute-lisp-command)
              (sense-func ((!t) lisp-true?))
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (error-func null-function)
              (translation (compare-length-sense ?l1 ?l2)))))


(defoperator GET-PATH ()
  (precond (satisfy (current.shell csh)))
  (effect (forall ((directory !d))
                  (exists ((path !p))
                          (when (current.path !d)
                            (and (observe (pathname !d !p))
                                 (observe (current.path !d)))))))
  (interface ((exec-func execute-unix-command)
      (translation ("echo $path"))
      (sensor-bind-func ucpop::my-get-sense-bindings)
```

```
      (sense-func ((!d !p) (get-path-sense :new))))))

(defoperator ADD-PATH ((directory ?dir))
  (declare ((path ?dirname)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (pathname ?dir ?dirname))
(satisfy (current.path ?dir) f)))
  (effect (cause (current.path ?dir)))
  (interface ((exec-func execute-unix-command)
      (error-func default-unix-error?)
      (translation ("set path = ( " ?dirname " $path )")))))

(defoperator RESET-PATH ()
  (precond (satisfy (current.shell csh)))
  (effect (and (forall ((directory !path))
                       (when (current.path !path)
                         (cause (current.path !path) F)))
       (cause (current.path.reset) t)))
  (interface ((exec-func execute-unix-command)
      (translation ("set path = (/bin)")))))
```

B.2 Person Operators

```
(defoperator netfind ((person ?person))
```

```
  (declare ((machine ?server)
    (machine.name ?server-name)
    (number ?time) (number ?current-time) (bool !busy)
    (number ?now)
            (city ?city) (field ?field) (state ?state)
            (country ?country)
            (organization ?organization) (person ?person)
            (domain !domain) (string ?key1)
            (string ?key2) (string ?key3)
            (userid !userid) (firstname ?first)
            (lastname ?last) (first.initial ?fi)
            (organization ?affiliation)))
  (precond (and (satisfy (current.shell csh))
(satisfy (machine.netfind.server ?server))
(satisfy (machine.name ?server ?server-name))
(or (contemplate (server.busy ?server? ?time) u)
    (and
     (contemplate (server.busy ?server ?time))
     (contemplate (seconds-ago ?time 300))))))
  (effect (and
           (when (and (eq !busy (bool t))
                      (contemplate (universal-time ?current-time)))
             (observe (server.busy ?server ?current-time)))
           (when (eq !busy (bool f))
             (and (observe (server.busy ?server ?time) u)
                  (when (and (neq !userid (userid "notfound"))
                             (or (city ?person ?key1)
                                 (organization ?person ?key1)
```

```
                (affiliation ?person ?key1))
            (or (country ?person ?key2)
                (state ?person ?key2)
                (eq ?key2 ""))
            (or (field ?person ?key3)
                (eq ?key3 "")))
  (and
   (when (and (person.domain ?person !domain)
              (userid ?person !userid !domain))
     (and
      (when (city ?person ?key1)
        (observe (city ?person ?city)))
      (when (organization ?person ?key1)
        (observe (organization ?person ?organization)))
      (when (affiliation ?person ?key1)
        (affiliation ?person ?affiliation))

      (when (country ?person ?key2)
        (observe (country ?person ?country)))
      (when (state ?person ?key2)
        (observe (state ?person ?state)))

      (when (field ?person ?key3)
        (observe (field ?person ?field)))

      (when (first.initial ?person ?fi)
        (observe (first.initial ?person ?fi)))
      (when (firstname ?person ?first)
```

```
                         (observe (firstname ?person ?first)))
                       (when (lastname ?person ?last)
                         (observe (lastname ?person ?last)))))))))))
  (interface ((exec-func execute-unix-command)
      (sense-func (!busy !userid !domain
                                (netfind-sense ?first ?last)))
      (logical-sense-call t)
      (translation ("netfind.exp " ?server-name " '"
                            ?last " " ?key1 " " ?key2 " "
                            ?key3  "'")))))


(defoperator WHOAMI ((userid !userid))
  (precond (satisfy (current.shell csh)))
  (effect (and (observe (current.userid !userid))))
  (interface ((exec-func execute-unix-command)
              (sensor-bind-func ucpop::my-get-sense-bindings)
              (sense-func ((!userid) bind-output-to-string))
              (translation ("whoami")))))


(defoperator FINGER ((string ?string))
  (declare ((machine ?machine)
                              (machine.name ?name) (domain ?domain)
       (bool !unfingerable)))
  (precond (and (satisfy (current.shell csh))
```

```
(satisfy (domain.machine.name ?domain ?name))
(satisfy (machine.name ?machine ?name))
(satisfy (machine.alive ?machine?))))
  (effect
   (and
    (observe (machine.unfingerable ?machine) !unfingerable)
    (forall ((person !person))
            (exists
       ((firstname !firstname) (lastname !lastname)
                (userid !userid) (first.initial !fi)
(finger.record !record))
             (when (and (eq !unfingerable (bool u))
                        (person.domain !person ?domain)
                        (or (lastname !person ?string)
                            (firstname !person ?string)
                            (and (userid !person ?string ?domain)
                                 (eq ?string !userid))))
               (and
                (observe (userid !person !userid ?domain))
                (observe (first.initial !person !fi))
                (observe (person.domain !person ?domain))
                (observe (firstname !person !firstname))
                (observe (lastname !person !lastname))
                (observe (finger.record !person !record
                                       ?domain))))))))
  (interface ((exec-func execute-unix-command-downcase)
              (sense-func (!unfingerable
                           (!person !userid !firstname
```

```
                                   !lastname !record !fi)
                          (finger-sense ?string :new)))
             (logical-sense-call t)
             (translation ("finger " ?string "@" ?domain)))))



(defoperator FINGER-MACHINE-GREP-USERID ((userid ?userid)
 (machine ?machine)
                                        (bool !finger-t)
 (number !idle-time)
                                        (bool !is-active))
  (declare ((machine.name ?machine-name)))
  (precond (and (satisfy (current.shell csh))
               (satisfy (machine.name ?machine? ?machine-name))
(satisfy (machine.alive ?machine?))))
  (effect (and (observe (logged.on ?userid ?machine) !finger-t)
       (observe (idle.time ?userid ?machine !idle-time))
       (observe (active.on ?userid ?machine) !is-active)))
  (interface ((exec-func execute-unix-command)
      (sense-func ((!finger-t !idle-time !is-active)
   (finger-user-sense ?userid)))
             (translation ("finger @" ?machine-name
                                       " |& egrep '^"
                                       ?userid "|Login|User'")))))
```

```
(defoperator STAFFDIR ((lastname ?lastname))
  (precond (and (satisfy (current.shell csh))))
  (effect
   (and (observe (staffdir.record ?lastname))
(forall ((person !person))
           (exists
            ((domain !domain)
             (userid !userid)
             (machine !machine)
             (machine.name !machine-name)
             (string !room) (phone.number !phone)
             (title !title) (department !dept) (mailbox !mail)
             (phone.number !voice) (bool !t-user) (bool !t-room)
             (bool !t-dept) (bool !t-voice) (bool !t-mail)
             (bool !t-title) (bool !t-phone)
             (first.initial !fi)
             (firstname !firstname))
            (when
                (and (lastname !person ?lastname)
                     (staff.member !person !domain))
              (and
               (observe (state !person "Washington"))
               (observe (country !person "Usa"))
               (observe (city !person "Seattle"))
               (observe (firstname !person !firstname))
               (observe (first.initial !person !fi))
               (observe (lastname !person ?lastname))
               (when (eq !t-user (bool t))
```

```
                (and (observe (staff.member !person !domain))
                     (observe (person.domain !person !domain))
                     (observe (userid !person !userid !domain))))
              (when (eq !t-phone (bool t))
                (observe (office.phone !person !phone)))
              (when (eq !t-title (bool t))
                (observe (person.title !person !title)))
              (when (eq !t-room (bool t))
                (observe (office.room !person !room)))
              (when (eq !t-title (bool t))
                (observe (mailbox !person !mail)))
              (when (eq (!t-voice (bool t)))
                (observe (voicemailnum !person !voice) !t-voice))
              (when (eq (!t-dept (bool t)))
                (observe (department !person !dept) !t-dept)))))))

 (interface
  ((translation ("staffdir -full -N " ?lastname))
   (exec-func execute-unix-command)
   (error-func staffdir-error)
   (logical-sense-call t)
   (sense-func ((!person !firstname !userid
 !domain !t-user !room !t-room !phone !t-phone
 !title !t-title !dept !t-dept
 !mail !t-mail !voice !t-voice !fi)
(staffdir-sense :new ?lastname))))))
```

```
(defoperator PERSON-OFFICE-ROOM ((person ?person) (room.number !room))
  (declare ((userid ?userid) (domain ?domain)))
  (precond (and (satisfy (current.shell csh))
(satisfy (current.domain ?domain))
(satisfy (userid ?person ?userid ?domain?))))
  (effect (and (observe (office.room ?person !room))))
  (interface
   ((translation ("grep " ?userid "
                  /cse/student-affairs/mailing-lists/room-*.dis"))
    (exec-func execute-unix-command)
    (logical-sense-call t)
    (sense-func ((!room) (person-office-room-sense ?userid))))))


(defoperator USERS-IN-ROOM ((room.number ?room))
  (precond (and (satisfy (current.shell csh))))
  (effect (forall ((userid !uid))
                  (when (userid.room !uid ?room)
                    (observe (userid.room !uid ?room)))))
  (interface
   ((translation ("cat /cse/student-affairs/mailing-lists/room-"
                    ?room ".dis"))
    (exec-func execute-unix-command)
    (logical-sense-call t)
    (sense-func ((!uid) (users-in-room-sense))))))

(defoperator PEOPLE-IN-ROOM ((room.number ?room))
  (declare ((person ?person) (userid ?userid) (domain ?domain)))
```

```
  (precond (and
(satisfy (userid.room ?userid ?room?))
(satisfy (current.domain ?domain))
(satisfy (userid ?person ?userid? ?domain?))
(satisfy (person.domain ?person ?domain?))
))
  (effect (observe (office.room ?person ?room))))



(defoperator INFER-OFFICE-PHONE-FROM-FINGER-REC ((person ?person)
 (phone.number !phone)
 (bool !phone-t))
  (declare ((finger.record ?f-rec) (domain ?domain)))
  (precond (and (satisfy (finger.record ?person? ?f-rec ?domain))))
  (effect
   (observe (office.phone ?person !phone) !phone-t))
  (interface
   ((exec-func execute-lisp-command-no-vcs)
    (translation (stringify ?f-rec))
    (sense-func ((!phone !phone-t)
 (office-phone-from-finger-rec-sense ?f-rec))))))

(defoperator INFER-HOME-PHONE-FROM-FINGER-REC ((person ?person)
       (phone.number !phone)
       (bool !phone-t))
  (declare ((finger.record ?f-rec) (domain ?domain)))
  (precond (and
```

```
(satisfy (finger.record ?person? ?f-rec ?domain))))
  (effect
   (observe (home.phone ?person !phone) !phone-t))
  (interface
   ((exec-func execute-lisp-command-no-vcs)
    (translation (stringify ?f-rec))
    (sense-func ((!phone !phone-t)
 (home-phone-from-finger-rec-sense ?f-rec))))))


(defoperator INFER-OFFICE-PHONE-SHARED ((person ?person)
                                        (phone.number ?phone))
  (declare ((person ?officemate) (room.number ?shared-room)))
  (precond
   (and (neq ?person ?officemate)
(satisfy (office.room ?person ?shared-room))
(satisfy (office.room ?officemate? ?shared-room?))
(satisfy (office.phone ?officemate? ?phone))))
  (effect
   (observe (office.phone ?person ?phone))))


(defoperator userid-login-machines ((machine ?machine)
    (machine !machine))
  (declare ((userid ?userid) (machine.name ?name)))
  (precond (and
     (satisfy (machine.name ?machine? ?name))
     (satisfy (machine.alive ?machine?))
```

```
    (satisfy (machine.domain ?machine? "cs.washington.edu"))))
  (effect (forall ((machine !machine))
             (exists
              ((machine.name !name))
              (when (userid.login.machine ?userid ?machine !machine)
                (and
                 (observe (userid.login.machine ?userid
                                                ?machine !machine))
                 (observe (machine.name !machine !name)))))))
  (interface
   ((translation
     ("rsh " ?name " \"who /var/adm/wtmp | grep "
             ?userid " | tail -50\""))
    (exec-func execute-unix-command)
    (logical-sense-call t)
    (sense-func ((!machine !name)
 (sense-userid-login-machines ?userid :new))))))
```

B.3 Machine Operators

```
(defoperator PING ((string ?machine-name)
                   (machine !machine))
  (declare ((machine.nickname !nickname) (domain !domain) (bool !tv)
                                          (machine.name !name)))
  (precond (satisfy (current.shell csh)))
  (effect
```

```
   (when (or (and (machine.name !machine ?machine-name)
                  (domain.machine.name !domain !name))
             (and (machine.domain !machine !domain)
                  (current.domain !domain)
                  (domain.name.nickname !domain !name !nickname)
                  (machine.nickname !machine ?machine-name)))
     (and (observe (machine.alive !machine) !tv)
          (when (machine.alive !machine)
            (and
             (observe (machine.name !machine !name))
             (observe (machine.nickname !machine !nickname))
             (observe (machine.domain !machine !domain)))))))
  (interface ((exec-func execute-unix-command)
      (translation ("/usr/etc/ping " ?machine-name))
      (logical-sense-call t)
      (sense-func ((!tv !machine !name !nickname !domain)
                            (ping-machine-sense :new ?machine-name))))))


(defoperator FINGER-MACHINE ((machine ?machine))
  (declare ((machine.name ?machine-name)))
  (precond (and (satisfy (current.shell csh))
(contemplate (is-bound ?machine))
(satisfy (machine.alive ?machine?))
(satisfy (machine.name ?machine? ?machine-name))
(contemplate (machine.unfingerable ?machine) u)))
  (effect (and
   (forall ((userid !uid))
```

```
                 (when (logged.on !uid ?machine)
                   (observe (logged.on !uid ?machine))))
   (forall ((userid !uid) (number !idle-time))
                 (when
                     (idle.time !uid ?machine !idle-time)
                   (observe (idle.time !uid ?machine !idle-time))))
   (forall ((userid !uid) )
                 (when (active.on !uid ?machine)
                   (observe (active.on !uid ?machine))))))
  (interface ((exec-func execute-unix-command)
             (sense-func (((!uid) (!uid !idle-time) (!uid))
   (finger-machine-sense ?machine)))
      (logical-sense-call t)
             (error-func finger-error)
      (translation ("finger @" ?machine-name)))))


(defoperator HOSTNAME ((machine !host))
  (declare ((machine.name !n) (domain ?domain-name)))
  (precond (and (satisfy (current.shell  csh))
(satisfy (current.domain ?domain-name))))
  (effect (when (current.host !host)
            (and (observe (machine.name !host !n))
                 (observe (current.host !host)))))
  (interface ((exec-func execute-unix-command)
             (sensor-bind-func ucpop::my-get-sense-bindings)
             (sense-func ((!host !n) (hostname-sense
                                       :new ?domain-name)))
```

```
             (translation ("hostname")))))


(defoperator UPTIME ((machine ?m1) (number !load)
                     (number !user-num))
  (declare ((machine.name ?n1)))
  (precond (and (satisfy (current.shell  csh))
               (satisfy (machine.name ?m1? ?n1))))
  (effect (and (observe (load ?m1 !load))
               (observe (users ?m1 !user-num))))
  (interface ((exec-func execute-unix-command)
             (sense-func ((!load !user-num) uptime-sense))
             (error-func null-function)
             (translation ("rsh " ?n1 " uptime")))))


(defoperator MACHINE ((string !type))
  (precond (satisfy (current.shell csh)))
  (effect (observe (current.machine.type !type)))
  (interface ((exec-func execute-unix-command)
             (sense-func ((!type) machine-sense))
             (translation ("machine")))))


(defoperator HINFO nil
  (declare ((domain ?domain)))
  (precond
   (and (satisfy (current.shell csh))
```

```
(satisfy (current.domain ?domain))))
  (effect
   (and
(forall ((machine !m))
            (exists
             ((machine.name !name)
              (machine.nickname !nickname) (domain ?domain)
              (bool !tv-xterm) (machine.model !model)
              (bool !unfingerable) (bool !unrshable)
              (string !room) (room.number !room-number))
             (when (machine.domain !m ?domain)
               (and
                (observe (machine.nickname !m !nickname))
                (observe (machine.xterm !m) !tv-xterm)
                (observe (machine.unrshable !m) !unrshable)
                (observe (machine.unfingerable !m) !unfingerable)
                (observe (machine.name !m !name))
                (observe (machine.domain !m ?domain))
                (observe (machine.room !m !room))
                (observe (machine.room.number !m !room-number))
                (observe (machine.model !m !model))
                (forall ((userid !admin))
                        (when (machine.admin !m !admin)
                          (observe (machine.admin !m !admin))))
                (forall ((op.sys !op-sys))
                        (when (machine.op.sys !m !op-sys)
                          (observe (machine.op.sys !m !op-sys))))
                (forall ((display.color !color))
```

```
                        (when (machine.color !m !color)
                          (observe (machine.color !m !color)))))))))
  (interface ((exec-func execute-unix-command)
       (sense-func ((!m !name !tv-xterm
!nickname !model !room !room-number
!unfingerable
(!admin) (!op-sys)
(!color))
    (hinfo-all-sense ?domain :new)))
      (logical-sense-call t)
      (sensor-bind-func ucpop::my-get-sense-bindings)
      (translation ("hinfo")))))
```

```
(defoperator FIND-DOMAIN ((machine ?machine) (domain !domain))
  (declare ((machine.name ?m-name)))
  (precond (satisfy (machine.name ?machine? ?m-name)))
  (effect (observe (machine.domain ?machine !domain)))
  (interface ((exec-func execute-lisp-command-no-vcs)
      (sense-func ((!domain) bind-output-to-string))
      (translation (find-domain-name ?m-name)))))
```

B.4 Printer Operators

```
(defoperator LPR ((printer ?printer) (simple.file ?file))
  (declare ((printer.name ?printer-name)
```

```
          (path ?pathname)
          (filename ?filename)
          (pstext ?ps)
          (dvi.document ?dvi)
          (document ?doc)
          (print.job !printout)
          (string ?option)
          (print.job ?printout)
          (string !jobname)
          (directory ?parent-dir)))
(precond (and (satisfy (current.shell  csh))
              (satisfy (pathname ?file ?pathname))
              (satisfy (filename ?file ?filename))
              (satisfy (printer.name ?printer ?printer-name))
              (satisfy (located.at ?doc ?pathname))
              (or (and (eq ?doc ?dvi) (eq ?option " -d "))
                  (and (eq ?doc ?ps) (eq ?option " ")))))
(effect (and (cause (printed ?file ?printer))
             (when (and (job.name ?printout !jobname)
                        (job.printer ?printout ?printer))
               (and (cause (document.job ?doc ?printout))
                    (when (and (iscolor ?printer) (iscolor ?doc))
                     (cause (iscolor ?printout)))))))
(interface ((exec-func execute-unix-command)
            (sense-func ((!jobname) (lpr-sense ?filename)))
            (translation ("lpr -h -P" ?printer-name
                                      ?option ?pathname)))))
```

```
(defoperator lpq (?printer)
  (declare ((printer.name ?printer-name)
            (print.job !job)
            (print.job !done-job)
            (string !job-name)
            (number !job-number)
            (thing !job-status)
            (userid !job-user)
            (status !job-status)
            (status !status)
            (status !queue-status)))
  (precond (and (satisfy (current.shell  csh))
                (satisfy (printer.name ?printer ?printer-name))))
  (effect (and (observe (print.queue.status ?printer !queue-status))
               (observe (printer.status ?printer !status))
               (forall ((print.job !job))
                  (when (and (job.printer !job ?printer)
                             (job.status !job working))
                    (and (observe (job.name !job !job-name))
                         (observe (job.number !job !job-number))
                         (observe (job.user !job !job-user))
                         (observe (job.printer !job ?printer))
                         (observe (job.status  !job working)))))))
  (interface ((exec-func execute-unix-command)
              (logical-sense-call t)
              (sense-func (!status !queue-status
                           (!job !job-number !job-name !job-user)
```

```
                                  (lpq-sense :new)))
             (translation ("lpq -P" ?printer-name)))))
```

B.5 Web Operators

```
(defoperator get-web-page ((url ?url) (http.text !text))
  (declare ((web.page ?www)))
  (precond (satisfy (current.shell csh)))
  (effect (when (located.at ?www ?url)
            (and (observe (contents ?www !text))
                 (observe (located.at ?www ?url)))))
  (interface ((exec-func execute-unix-command)
              (translation ("lynx -source " ?url))
              (logical-sense-call t)
              (sense-func (!text (output)))
              (error-func no-error)
              (terminate-detect read-unix-prompt))))
```

```
(defoperator get-urls-in-page ((web.page ?p))
  (declare ((http.text ?text)))
  (precond (and (satisfy (contents ?p? ?text))
                (satisfy (located.at ?p ?pu))))
  (effect (forall ((web.page !child))
             (exists ((url !u) (string !name))
                (when (points.to ?p !child)
```

```
                  (and (observe (points.to ?p !child))
                       (observe (name !child !name))
                       (observe (located.at !child !u)))))))
  (interface ((exec-func execute-lisp-command-no-vcs)
              (translation (identity ?text))
              (logical-sense-call t)
              (sense-func ((!child !u !name)
                           (scan-for-urls :new ?pu)))
              (error-func no-error)
              (terminate-detect read-unix-prompt))))


(defoperator web-ftp-file-info ((ftp.directory !d))
  (declare ((url ?url)))
  (precond (and (satisfy (current.shell csh))))
  (effect (when (located.at !d ?url)
            (and (observe (located.at !d ?url))
                 (forall ((ftp.file !f))
                    (exists ((url !u)
                        (minute !minute)
                        (hour !hour)
                        (date !day)
                        (month !month)
                        (year !year)
                        (time !t)
                        (filename !n))
                       (when (points.to !d !f)
                         (and
```

```
                         (observe (points.to !d !f))
                         (observe (located.at !f !u))
                         (observe (creation.date !f !t))
                         (observe (month !t !month))
                         (observe (date !t !day))
                         (observe (year !t !year))
                         (observe (hour !t !hour))
                         (observe (minute !t !minute))
                         (observe (name !f !n)))))))))
  (interface ((exec-func execute-unix-command)
              (translation ("lynx -source " ?url))
              (logical-sense-call t)
              (sense-func (!d (!t !month !year
                                   !day !hour !minute !u !n !f)
                             (ftp-sense :new :new :new)))
              (error-func no-error)
              (terminate-detect read-unix-prompt))))


(defoperator save-web-page ((document ?f)
                            (directory ?dir))
  (declare ((url ?url)
            (directory ?dir)
            (path ?path)
            (path ?dirname)
            (filename ?n)
            (file ?random)))
  (precond (and (satisfy (located.at ?f? ?url))
```

```
                (satisfy (pathname ?dir ?dirname))
                (satisfy (name ?f ?n))
                (satisfy (concat ?dirname? "/" ?n? ?path))))
  (effect (exists ((file !local))
             (and (cause (copy.of ?f !local ?dir))
                  (cause (parent.directory !local ?dir))
                  (cause (pathname !local ?path))
                  (cause (filename !local ?n)))))
  (interface ((exec-func execute-unix-command)
      (translation ("lynx -source " ?url " >! " ?path))
              (sense-func ((!local) (ident :new))))))
```

B.6 Display Operators

```
(defoperator xless ((path ?filepath))
  (documentation "Xless displays text files")
  (declare ((simple.file ?file) (text.document ?doc)))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))
))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func default-unix-error?)
      (translation ("xless " ?filepath))
      ))
  )
```

```
(defoperator ghostview ((path ?filepath))
  (documentation "Ghostview displays Postscript Files")
  (declare ((simple.file ?file) (postscript.document ?doc)))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))
))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func no-error)
      (translation ("ghostview " ?filepath))
      ))
  )


(defoperator xv ((path ?filepath))
  (documentation "XV displays some graphical image type files")
  (declare ((simple.file ?file) (image.document ?doc)))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func default-unix-error?)
      (translation ("xv " ?filepath))
      ))
  )
```

```
(defoperator xdvi ((path ?filepath))
  (documentation "Xdvi displays DVI files from TeX")
  (declare ((simple.file ?file) (dvi.document ?doc)))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))
))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func default-unix-error?)
      (translation ("xdvi " ?filepath))
      ))
  )
```

```
(defoperator mpeg.play ((path ?filepath))
  (documentation "Mpeg_play displays mpeg format animation files")
  (declare ((simple.file ?file) (animation.document ?doc)))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))
))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func mpeg.play-error)
      (translation ("mpeg_play " ?filepath " ; echo \newline"))
      ))
  )
```

```
(defoperator netscape ((string ?filepath))
  (documentation "Display document using Netscape")
  (declare ((simple.file ?file) (web.page ?doc)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (located.at ?doc ?filepath))
                ))
  (effect (cause (displayed ?doc)))
  (interface ((exec-func execute-unix-command)
      (error-func default-unix-error?)
      (translation ("netscape -remote \"openURL("
                            ?filepath ")\"" ))
      ))
  )

(defoperator file-type ((path ?filepath))
  (documentation "Find out type of document contained in file")
  (declare ((simple.file ?file) (document !doc) (string ?type)))
  (precond (and (satisfy (current.shell csh))
                (satisfy (pathname ?file ?filepath))))
  (effect (observe (located.at !doc ?filepath)))
  (interface ((exec-func execute-unix-command)
              (error-func find-file-type-error)
              (logical-sense-call t)
              (translation ("file " ?filepath))
      (sensor-bind-func xii::my-get-sense-bindings)
              (sense-func ((!doc) (file-sense ?filepath :new)))
              (error-func no-error)
```

```
      )))


(defoperator latex ((path ?filepath))
  (documentation "Converts latex files to dvi files")
  (declare ((simple.file !dvifile)
    (simple.file ?file)
    (path ?dvipath)
    (directory ?dir)
    (tex.document ?doc)
            (dvi.document !dvidoc)
    ))
  (precond (and (satisfy (string.in.file "\batchmode" ?file?))
(satisfy (located.at ?doc ?filepath))
(satisfy (pathname ?file ?filepath))
(satisfy (latexed.path ?filepath ?dvipath))
))
  (effect (and (cause (pathname !dvifile ?dvipath))
               (cause (located.at !dvidoc ?dvipath))
               (cause (derived.from !dvidoc ?doc))
               (when (title ?doc ?title)
                 (cause (title !dvidoc ?title)))
               (when (author ?doc ?author)
                 (cause (author !dvidoc ?author)))
       ))
  (interface ((exec-func execute-unix-command)
      (sensor-bind-func ucpop::my-get-sense-bindings)
            (sense-func ((!dvifile !dvidoc)
```

```
                              (converter-sense :new :new)))
      (error-func default-unix-error?)
      (translation ("latex2e " ?filepath))
      ))
  )


(defoperator dvips ((path ?filepath))
  (documentation "Converts latex dvi files to postscript")
  (declare ((simple.file !psfile)
    (simple.file ?file)
    (path ?pspath)
    (directory ?dir)
            (dvi.document ?doc)
            (postscript.document !psdoc)
    ))
  (precond (and (satisfy (current.shell csh))
(satisfy (located.at ?doc ?filepath))
(satisfy (pathname ?file ?filepath))
(sense (dvitps.path ?filepath ?pspath))
(satisfy (file.type ?file "data"))
))
  (effect (and (cause (pathname !psfile ?pspath))
               (cause (located.at !psdoc ?pspath))
               (cause (derived.from !psdoc ?doc))
               (when (title ?doc ?title)
                 (cause (title !psdoc ?title)))
               (when (author ?doc ?author)
```

```
                (cause (author !psdoc ?author)))
      ))
 (interface ((exec-func execute-unix-command)
     (sensor-bind-func ucpop::my-get-sense-bindings)
             (sense-func ((!psfile !psdoc)
                             (converter-sense :new :new)))
     (error-func default-unix-error?)
     (translation ("dvitps -o" ?pspath " " ?filepath))
     ))
 )
```