

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
800-521-0600

UMI<sup>®</sup>



# **Diagnosis and Planning With Resource Constraints**

by

James A. Kurien

B. S., Rensselaer Polytechnic Institute, 1989

Sc. M., Rensselaer Polytechnic Institute, 1994

Sc. M., Brown University, 1995

Thesis

Submitted in partial fulfillment of the requirements  
for the Degree of Doctor of Philosophy  
in the Department of  
Computer Science at Brown University

May 2003

UMI Number: 3087292



---

UMI Microform 3087292

Copyright 2003 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

Copyright  
by  
James A. Kurien  
2003


This dissertation by James A. Kurien  
is accepted in its present form by the Department of Computer Science  
as satisfying the dissertation requirement  
for the degree of Doctor of Philosophy

Date 4/15/03

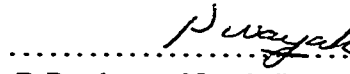
  
Prof. Leslie Pack Kaelbling, Ph.D., Advisor

Recommended to the Graduate Council


Date 4/9/03

  
Tom L. Dean, Ph.D., Reader

Date 4/8/03


  
P. Pandurang Nayak, Ph.D., Reader

Date 4/7/03

  
David E. Smith, Ph.D., Reader

Approved by the Graduate Council

Date 4/18/03

  
Karen Newman, Ph.D.  
Dean of the Graduate School

## Vita

James Kurien was born in Manchester, Connecticut on January 9, 1967. He received a Bachelors degree in computer science with a minor in applied mathematics from Rensselaer Polytechnic Institute in 1989, a Masters degree in computer science from Rensselaer in 1994, another Masters degree in computer science from Brown University in 1995, and a Ph.D. in Computer Science from Brown University in 2003. A believer in the value of life-long education, he graduates from California's traffic school system almost yearly.

James has been extremely fortunate to be employed at a number of fun and exciting research labs. While completing his first Masters degree, he began employment at the IBM T. J. Watson research lab. Thus began a trend. After completing the course work and preliminary examinations of his Ph.D. program, he began employment at the NASA Ames Research Center in California. Here he was a major contributor to the Remote Agent team that in 1999 demonstrated autonomous, AI-based operation of a spacecraft in deep space. This work received the 1999 NASA Software of the Year award and served as the inspiration for his thesis dissertation, on the diagnosis and control of complex physical systems such as spacecraft. In 2001, he was recruited by the Palo Alto Research Center, where he continued his thesis research and expanded his research interests to include hybrid diagnosis and distributed sensing.

James has published widely in the areas of model-based diagnosis, automated planning, partially observable Markov decision processes, and robotics. He has three patents pending, and serves on the Program Committee for the International Workshop on Principles of Diagnosis. His favorite Monkee is Peter Tork.

## Acknowledgments

For Abe, Ruth, Sonya, Matt and Mary.

I would have been extremely fortunate to have one person as kind, talented and patient as Leslie, Pandu, Dave or Tom take me on as an apprentice, let alone four.

I can barely remember being a withdrawn, defensive young man, employed stacking boxes in a dusty warehouse. How that person became open, joyful, and capable of earning a Ph.D. is a long story, but it never would have happened without the kindness and support of Ron Schack, Marie Simmons, David Ferrucci, Carolyn St. Jean, Myra Cordova, Dan Clancy, Mike Depumpo, Jon Rogers, Dan Mulligan and many others.

I would like to thank the Palo Alto Research Center for hiring me while my thesis was not yet complete and allowing me to dedicate a significant portion of my work schedule to completing it. I would like to thank my colleagues at PARC, especially Johan de Kleer, Feng Zhao, and Xenofon Koutsoukos, for their patience and support.

I would like to thank everyone with whom I worked on Remote Agent, Livingstone, and L2, at NASA Ames Research Center, Kennedy Space Center, Johnson Space Center and the Jet Propulsion Laboratories. Funding for a portion of this work was provided by the National Aeronautics and Space Administration, through the Cross-Enterprise Technology Development Program.

Brought to you by the letter W.



# Contents

<b>Vita</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>Table of Contents</b>	<b>viii</b>
<b>List of Illustrations</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Impact of Control Systems . . . . .	1
1.2 Problems in Machine Regulation at NASA . . . . .	5
1.3 Model-based Discrete Control Systems . . . . .	10
1.4 Overview of State Estimation . . . . .	12
1.5 Overview of Planning . . . . .	17
1.6 A Model-based Health Maintenance Capability . . . . .	19
1.7 Document Overview . . . . .	21
<b>2 Related Work in State Identification</b>	<b>23</b>
2.1 Partially Observable Markov Decision Processes . . . . .	24
2.2 Model-Based Diagnosis . . . . .	27
<b>3 Trajectory Identification</b>	<b>31</b>
3.1 Introduction . . . . .	31
3.2 Transition systems . . . . .	33

3.3	Trajectory Identification . . . . .	36
3.4	Infinitesimals . . . . .	37
3.5	Correspondence to the POMDP Formulation . . . . .	38
<b>4</b>	<b>Trajectory Tracking Algorithms</b>	<b>40</b>
4.1	The <i>CBFS-track</i> Trajectory Tracking Algorithm . . . . .	40
4.2	The <i>Livingstone</i> Algorithm . . . . .	43
4.3	The Conflict Coverage Algorithm . . . . .	45
4.4	Additional Related Work . . . . .	46
<b>5</b>	<b>Decreasing the problem size</b>	<b>48</b>
5.1	Restricting $\mathcal{M}_{\mathcal{T}}$ . . . . .	49
5.2	Eliminating intermediate observations . . . . .	50
5.3	Selective Model Extension . . . . .	51
5.4	Finite Horizons . . . . .	53
<b>6</b>	<b>Results for Diagnosis</b>	<b>57</b>
6.1	Validation By NASA Spacecraft Engineers . . . . .	57
6.2	Experiments . . . . .	60
6.2.1	Valves . . . . .	61
6.2.2	CB and ISPP . . . . .	66
<b>7</b>	<b>New Approaches to Conformant Planning</b>	<b>74</b>
7.1	Introduction . . . . .	74
7.2	A Fragment-based Conformant Planner . . . . .	77
7.3	Search Strategies for Fragment-based Planning . . . . .	84
7.4	Implementation Using a SAT Planner . . . . .	88
7.5	A Conflict-based Conformant Planner . . . . .	90
7.6	An Extension to Handle Conditional Actions . . . . .	92
7.7	An Extension to Handle Ramifications . . . . .	96
7.8	A Brief Illustration . . . . .	99

7.9	Summary . . . . .	103
<b>8</b>	<b>Experimental Results for Conformant Planning</b>	<b>105</b>
8.1	Performance of the Fragment-based Planner . . . . .	105
8.1.1	Performance Comparison on the BTC Domain . . . . .	106
8.1.2	Performance with an exponential set of worlds . . . . .	111
8.2	Comparison of Search Strategies for fragPlan . . . . .	114
8.3	Performance on the Cassini Domain . . . . .	118
8.4	Performance of Conflict Planner . . . . .	126
8.5	Performance of A Conjunctive Conformant Planner . . . . .	129
<b>9</b>	<b>Safe, Conformant Planning with Optimization</b>	<b>132</b>
9.1	Motivation . . . . .	132
9.2	Optimally Safe Conformant Planning . . . . .	133
9.2.1	Safety . . . . .	133
9.2.2	Optimal Plans . . . . .	135
9.3	SCOPE algorithms . . . . .	137
9.4	Partial Ordering of Constraints . . . . .	138
9.5	Optimization Via Relaxing The Problem Scope . . . . .	140
9.5.1	Computing <i>Difficulty</i> for Goals and Safety . . . . .	143
<b>10</b>	<b>Experimental Results for SCOPE</b>	<b>145</b>
10.1	Introduction . . . . .	145
10.2	Overview of SCOPE Results . . . . .	145
10.3	Observations on SCOPE and fragPlan . . . . .	152
10.4	SCOPE Performance When No Conformant Plan Exists . . . . .	160
<b>11</b>	<b>Related Work in Acting Under Uncertainty</b>	<b>164</b>
11.1	Belief Replanning . . . . .	165
11.2	Conformant Planning . . . . .	166
11.3	Decision Theoretic Planners . . . . .	167

11.4 MDP-based Heuristics . . . . .	168
<b>12 Conclusions</b>	<b>170</b>
12.1 Contributions . . . . .	170
12.2 Future Work . . . . .	171
12.3 Living With Failure . . . . .	173

## List of Figures

1.1	Typical Control System Schematic . . . . .	3
1.2	Simple propulsion system . . . . .	6
1.3	Cassini propulsion system schematic. . . . .	7
1.4	The Valve Driver Example . . . . .	11
1.5	The Automaton Representing a Valve . . . . .	11
1.6	An Influence Diagram for the Valve . . . . .	13
1.7	An Influence Diagram for the VDU, Two Valves and a Tank. Some arcs not shown for clarity. . . . .	14
1.8	An Influence Diagram for the VDU System over 4 Time Steps . . . . .	15
1.9	Simplified propulsion system schematic. . . . .	16
1.10	Model-based Health Maintenance . . . . .	20
2.1	Propulsion system schematic. . . . .	26
2.2	Evolution of a Valve Driver Unit and Valves . . . . .	29
3.1	A Simplified Valve Automaton . . . . .	32
3.2	Simplified Valve Automaton with $\tau$ Variables . . . . .	33
4.1	Evolution of the VDU/valve system . . . . .	41
4.2	CBFS-based trajectory tracking algorithm . . . . .	41
4.3	Two evolutions of the system . . . . .	42
4.4	Livingstone trajectory tracking algorithm . . . . .	44
4.5	Conflict Coverage Tracking Procedure . . . . .	47

5.1	Evolution before commanding the valves . . . . .	49
5.2	Evolution upon commanding the valves . . . . .	49
5.3	Expansion of the VDU Problem to Depth $m = 3$ . . . . .	53
5.4	Summarization of the initial trajectory, $\{\tau_{vdu,0}=\text{Hang}\}$ . . . . .	54
5.5	Summarization of the initial trajectory $\{\tau_{v1,1}=\text{Stick}, \tau_{v2,1}=\text{Stick}\}$ . . . . .	55
5.6	The Complete Representation . . . . .	56
6.1	The X-37 Vehicle . . . . .	58
6.2	The X-34 Vehicle . . . . .	58
6.3	Typical L2 Deployment Architecture . . . . .	60
6.4	X-34 Model Schematic . . . . .	61
6.5	Schematic for the X34 LOX Subsystem . . . . .	62
6.6	Schematic for the X34 Fuel Subsystem . . . . .	63
6.7	X-34 Diagnosis Scenarios . . . . .	64
6.8	Evolution of a Valve Driver Unit and Valves . . . . .	64
6.9	The Circuit Breaker (CB) Model . . . . .	66
6.10	The In-Situ Propellant Production (ISPP) Model . . . . .	67
6.11	CB - Single Failure After 598 Steps . . . . .	70
6.12	ISPP - Independent failures at steps 27, 32 33 . . . . .	70
6.13	ISPP - 4 Identical failures over 27 Steps . . . . .	71
6.14	CB - 39 Identical failures over 618 Steps . . . . .	72
7.1	Cassini Propulsion System Schematic . . . . .	75
7.2	A Simplistic Valve System . . . . .	78
7.3	A Plan For 6 Valves in A Row . . . . .	78
7.4	Generating A Plan for Restoring Flow When One of Three Valves is Closed . . . . .	79
7.5	Simple Fragment Planner . . . . .	82
7.6	A Recursive, Complete Fragment Planner . . . . .	83
7.7	Flexible Fragment Planner . . . . .	85
7.8	A Simple Fragment-based Plan That Failed . . . . .	86

7.9	A Conflict-Based Conformant Planner . . . . .	91
7.10	Generating A Plan With Aspects . . . . .	96
7.11	Cassini Propulsion System Schematic . . . . .	97
7.12	A Minimal Plan When M1's Valves are Stuck . . . . .	100
7.13	Iterations Required for Two Valves On One Engine . . . . .	101
7.14	Iterations Required for Two Valves On One Engine, Sorted . . . . .	102
7.15	Iterations Required for One Valves On Each Engine . . . . .	103
7.16	Iterations Required With Sufficient Time Steps . . . . .	103
8.1	A bomb in the toilet plan for 6 packages, 1 toilet . . . . .	106
8.2	A parallel bomb in the toilet plan for 6 packages, 2 toilets . . . . .	106
8.3	A serial bomb in the toilet plan for 6 packages, 1 toilet . . . . .	107
8.4	<i>fragPlan</i> and serial planners on BTC . . . . .	107
8.5	Time versus parallelism on BTC for <i>fragPlan</i> . . . . .	109
8.6	<i>fragPlan</i> and <i>C-plan</i> on BTC . . . . .	110
8.7	Effect of Worlds on Iterations Per World for <i>fragPlan</i> . . . . .	112
8.8	Effect of Worlds on <i>fragPlan</i> Search Time . . . . .	113
8.9	Probing vs. Bubbling on Asymmetric Worlds . . . . .	115
8.10	Effect of Time Steps on <i>fragPlan</i> . . . . .	116
8.11	Cassini Propulsion System Schematic . . . . .	117
8.12	Performance of 31 trials of <i>fragPlan</i> on 3 Cassini problems . . . . .	117
8.13	Performance of 31 trials of <i>fragPlan</i> on 4 Cassini problems . . . . .	118
8.14	Effect of Time Steps on <i>fragPlan</i> time for the Cassini domain . . . . .	120
8.15	Effect of Time Steps on <i>fragPlan</i> iterations for the Cassini domain . . . . .	120
8.16	Runtime of <i>fragPlan</i> on Cassini with total time<200 and SAT limit<250 . .	123
8.17	Iterations of <i>fragPlan</i> on Cassini with total time<200 and SAT limit<250 .	123
8.18	Runtime of <i>fragPlan</i> on Cassini with total time<200 and SAT limit<20 . .	125
8.19	Iterations of <i>fragPlan</i> on Cassini with total time<200 and SAT limit<20 . .	125
8.20	Effect of Time Steps on <i>fragPlan</i> time for the Cassini domain . . . . .	125
8.21	A Plan For A Single World . . . . .	127

8.22	A Simple Fragment-based Plan That Failed . . . . .	128
8.23	Bomb in the Toilet Performance, Limited to 15 Seconds Per Attempt . . . .	129
8.24	Performance of the Conjunctive Planner . . . . .	130
9.1	The Outline of a SCOPE Planning Algorithm . . . . .	138
9.2	addPlan . . . . .	139
9.3	dropPlan . . . . .	140
9.4	A More Complete Planning Framework . . . . .	142
9.5	Unit Propagation With Support Tracking . . . . .	144
10.1	An Intuition of Planning Time Vs. Constraints . . . . .	146
10.2	Four control schemes for SCOPE . . . . .	147
10.3	Four Simple SCOPE Strategies . . . . .	147
10.4	Typical Performance for Four Simple SCOPE Strategies . . . . .	150
10.5	SCOPE on BTC, 10 Packages, 22 Time Steps, Part 1 . . . . .	153
10.6	SCOPE on BTC, 10 Packages, 22 Time Steps, Part 2 . . . . .	154
10.7	SCOPE on Modified Ring World of Size 5 . . . . .	155
10.8	SCOPE on the Linear Logistics Problem of Length 6 . . . . .	156
10.9	Expected Performance versus Time for fragPlan and SCOPE . . . . .	158
10.10	Plans found by SCOPE compared to those found by fragPlan . . . . .	159
10.11	SCOPE when resources set a maximum number of solvable worlds . . . . .	161
10.12	Performance of dropPlan versus horizon for several run times . . . . .	162
10.13	Performance of addPlan and dropPlan versus planning horizon . . . . .	162



# **Chapter 1**

## **Introduction**

This work concerns the automatic control of complex physical systems such as spacecraft or life support systems even in the face of equipment failures or other unexpected events. The first section of this introductory chapter provides a high level introduction to the concept of a control system. The second section motivates why additional research is needed when very successful control systems have been developed for everything from automobile engines to cruise missiles to Furby dolls. The third section briefly introduces the subject of this work, a type of control system that attempts to achieve robustness by performing a significant amount of reasoning about the physical device it controls. The final section introduces the remaining chapters of the document.

### **1.1 The Impact of Control Systems**

Over the past few decades, many common machines have slowly evolved into marvels of functionality, efficiency and reliability. At the same time they have, to the casual observer, kept their familiar forms. Passenger aircraft, looking much like the military transports from which they were derived a half century ago, routinely fly thousands of miles over oceans on autopilot with four, and now only two, engines. Catastrophic failure rates measure once in millions of flights. Automotive disc brakes, first developed in the 1890's, now bring automobiles to a halt on ice or gravel without loss of control and without any special skill on the part of the driver. Internal combustion engines, first used in the mid-nineteenth century, can now power an automobile for a decade without adjustment, with significantly

greater power and less pollution than was possible in the recent past. In order to explore one of the factors driving this evolution, let's consider how automobile engines evolved between 1965 and 1995.

An internal combustion engine repeatedly draws air and fuel into an internal chamber where a spark is applied, producing power from the resulting small explosions. While reliable for its day, an engine built in 1965 is a temperamental beast compared to its modern brethren. It's prone to hard starting at extreme temperatures, requires adjustment every few thousand miles, and is apt to spew unburnt fuel and other pollutants from its exhaust. If clogged filters, component failures or deliberate modifications create minor alterations in the way air or fuel are delivered to the engine, it loses power or ceases functioning, leaving the owner to divine what needs to be replaced or adjusted. A 1995 engine starts immediately regardless of conditions, goes 100,000 miles without adjustment, and produces an order of magnitude less pollution than its predecessor. The 1995 engine is impervious to any reasonable change in how fuel and air are delivered. If component failures prevent smooth power production, the engine can often enter a "limp home" mode that reliably produces minimal power, and report the cause of the problem to the user. This revolution in efficiency, reliability and robustness may not seem surprising given how the world can change in three decades. What is interesting to note is that the vast majority, of the components that make up the 1965 Ford engine the author has in mind, perhaps some 80% by weight, would be indistinguishable to the casual observer from the corresponding components of the 1995 derivative. Many of the parts are in fact interchangeable. The sole significant difference between these two machines lies in their *control systems*.

As illustrated in Figure 1.1, the purpose of a closed-loop control system is to receive observations from a physical system, estimate the current state of the system, and take actions that move the system to a desired state of operation. This general description of the problem can be applied whether one is attempting to run an engine efficiently, land an airplane automatically, or regulate the human heart with a pacemaker. In the case of the automobile engine, the control system must determine how much fuel to add to the air that is entering the engine and decide at exactly what point in time to apply the spark

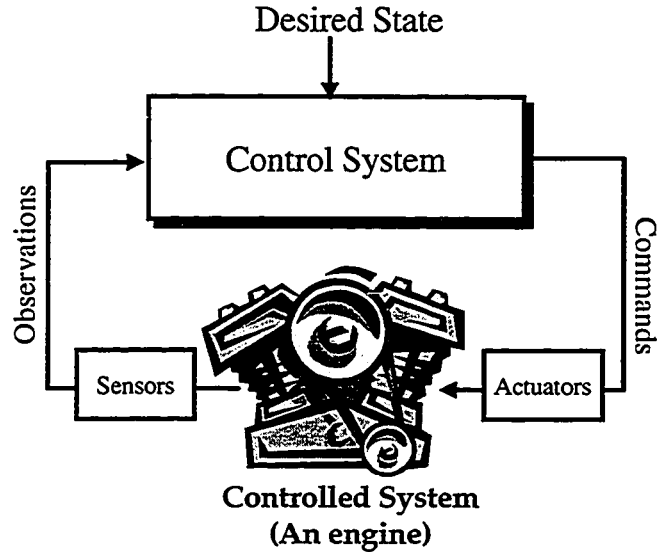


Figure 1.1: Typical Control System Schematic

to keep the process running smoothly. A control system may be extremely simple. In older home thermostats, a metal spring expands or contracts with variation in temperature and an attached switch turns the house's heater on or off. A control system may also be an extremely complex affair, wherein thousands of measurements are used by a team of humans and computers to determine what action to take to control a complex process such as refueling the space shuttle.

Conceptually, the difference between the 1965 and 1995 engines is to what extent the control system of each is aware of the engine's conditions and how flexibly it can respond. The control system for the 1965 engine is designed to virtually eliminate the need for on-board decision making. When the driver depresses the accelerator pedal, a flap opens to allow more air to rush into the engine. The rushing air flows past a fuel source, drawing fuel with it. At the factory, a fuel opening is chosen such that the amount of air that usually rushes through the engine will draw enough fuel to provide adequate performance across a range of usual conditions. If the air flow, air density, or fuel flow change significantly, the opening no longer provides the right amount of fuel. The combustion process loses efficiency or simply stops. The driver then provides the necessary expertise to change the environment to suit the control system, perhaps by letting the car warm up before using it,

spraying starting fluid in the engine, or letting excess fuel evaporate.

In contrast, the 1995 engine is controlled by a software system that captures the expertise of Ford's control engineers. Sensors determine the temperature and mass of the air rushing into the engine for each firing. The engine temperature, barometric pressure, and a number of other measurements are also taken. Based upon these measurements, the control software running on a computer under the dashboard determines how much fuel is required for optimal combustion. It instructs fuel injectors in the engine to open just long enough to spray the desired amount of fuel into the engine, and similarly controls the moment at which the air/fuel mixture will be ignited. In the exhaust stream that results from combustion, oxygen sensors inform the computer whether the ratio of air to fuel being burned is correct. If too little or excess fuel is being delivered, either a sensor or the fuel injector must not be responding properly and commands to the injectors must be adjusted accordingly. Similarly, the computer continually monitors the output of all sensors for indications that a sensor may be malfunctioning. In this case, the output of the sensor is ignored and the computer does its best to operate the engine using only the remaining sensors. Then engine can even be ordered to run through a self test, wherein it changes the commands to the fuel injectors and sparking system and watches for the appropriate responses on the sensors in an attempt to single out problematic components. All failures, whether discovered during normal operations or active testing, are reported to the user and can be downloaded to a diagnostic system along with any anomalous sensor readings for further investigation.

This section was intended to suggest the following intuitions:

- *The job of a control system is to adjust a machine or physical process based upon an estimate of the current conditions of the process, in order to optimize performance.*
- *The estimation method and types of adjustments made in response may be very simple or quite complex.*
- *A machine can be made more robust to failures or environmental change if those changes are identified and the machine is adjusted based upon how the changes will effect its operation.*

- *Increasing the complexity of a machine, by adding sensors, actuators and control software, can paradoxically increase its robustness and the simplicity with which it is operated.*

## **1.2 Problems in Machine Regulation at NASA**

In this work, we will focus on situations in which the internal state of a machine must be estimated and controlled. NASA has an endless variety of problems involving the internal regulation of complex machines where the system must continue to operate even in the face of failures. These include operation of human life support systems for Earth-orbit and future missions to the moon and Mars, operation of automated propellant production systems on Mars to enable future exploration, and diagnosis and control of vehicles in the atmosphere, Earth orbit or deep space. Given these exciting and critical applications for control systems, an important question to ask is, why aren't existing methods for developing control systems adequate? The answer lies in the significant differences between these applications and those handled so successfully by industry.

Most importantly, the economics at NASA and a manufacturer such as Ford Motor Company are reversed. Ford released version one of its electronic engine control system in 1984, and by the time version five was released in 1995, it had been installed in tens of millions of relatively similar engines in Ford vehicles. A large amount of effort could be expended to develop the initial control system, as that cost would be amortized over many automobiles. Detailed analysis by engineers to improve performance or reduce per-unit cost could potentially be justified by the profit produced by selling millions of units. In addition, new versions of the system could be developed incrementally at a relatively slow rate, drawing upon the experience gained from running millions of copies of the system under varying conditions for several years. At NASA, the current practice is to develop each spacecraft design almost from scratch over a period of two to three years and produce only one or two copies of each design. The fully integrated spacecraft system will often be run for only a few hundred hours or not at all before being deployed in space. Thus our requirements upon the control system development process include:

- *Control systems must be developed cheaply and quickly in parallel with the hardware*

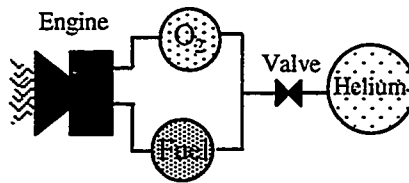


Figure 1.2: Simple propulsion system

*system.*

- *They must also be easy to modify once the fully integrated system is tested and deployed.*

A second important distinction is the range of failures over which the control system is expected to operate. If an automobile engine experiences a severe failure or a set of failures that was thought to be highly unlikely, the control system need not continue functioning. The driver can consult the owner's manual for a solution or the automobile can be towed to a shop and repaired. A small amount of down time over the life of an automobile is potentially acceptable, and understandable if a primary component fails. In contrast, many NASA systems such as deep space probes travel far beyond the reach of easy repair. If a component fails very early in a long mission, the control system must continue performing state estimation and control as best it can without that component. In addition, there are critical periods when a short down time will render useless a multi-year spacecraft mission costing hundreds of millions of dollars. For example, if a failure were to prevent a spacecraft from properly decelerating as it approaches a planet or other body it's attempting to orbit, it might burn up in the atmosphere or be flung uselessly into space. For spacecraft attempting to orbit the more distant planets, by the time mission controllers on Earth received a radio signal indicating that something was amiss, it would be too late to respond. Spacecraft must therefore carry their spare parts with them in the form of multiple copies of critical components (called *block redundancy*) or multiple methods for achieving the same control action using different components (called *functional redundancy*).

**Example 1** Consider the schematic of a simple, notional rocket propulsion system shown in Figure 1.2. The purpose of the system is to provide just the right amount of acceleration by combining fuel and an oxidizer in an engine for a specified amount of time. The helium

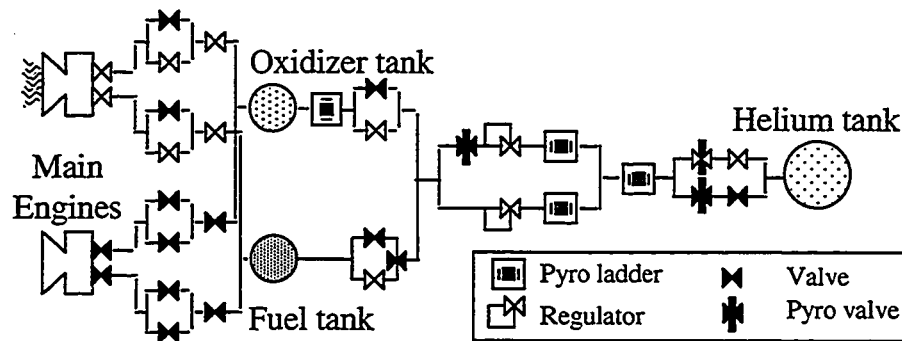


Figure 1.3: Cassini propulsion system schematic.

tank is filled with helium under high pressure. Conceptually, the control problem is quite simple. When the valve is opened, the high pressure normally forces oxygen and fuel into the engine where it is ignited to produce thrust. When sufficient thrust is achieved, the valve is closed. While this system is simple, it has the disadvantage that if any component fails, it ceases to operate.

Figure 1.3 illustrates the redundant propulsion system used in the Cassini spacecraft, designed to last a seven year cruise to Saturn and autonomously insert itself into orbit around Saturn. Two engines are provided in the case that one fails. Each engine is supplied with fuel and oxidizer through a complex arrangement of valves. Valves or pipe branches in parallel ensure that if valves stick closed, a redundant parallel valve can be used to allow fluid flow. Valves in series ensure that if valves stick open, an upstream valve can be closed to prevent fluid flow. Not shown are valve drivers that control the latch valves and a set of flow, pressure and acceleration sensors that provide partial observability of the system. There are approximately  $10^{15}$  possible configurations of the system including failures. Several hundred of those configurations produce thrust, depending upon which valves are open or closed. Given a set of failures, thrust configurations that can be reached without using a pyro valve are preferred, as pyro valves can only be opened or closed once. Regardless of the number of failures that occur, we'd like the control system to determine the current configuration of the propulsion system and find the best viable configuration of the system that will produce thrust.

The desire for robust operation over long periods without repair, and the resulting complex, redundant systems, introduce additional control system requirements:

- *Due to the number of combinations of failures that could occur over time, the control system must be able to control the system from a very large number of possible states.*
- *Due to the large number of possible states, the control system cannot explicitly record what action to take in each state.*
- *The control system must determine the best action, rather than simply a sufficient action, to take in order to reach the goal configuration.*
- *The control system must include discrete decisions (e.g., Should a valve be opened or closed?, and Should engine A or B be fired? )*
- *Because number of sensors is limited compared to the complexity of the system, the state of the system will not be directly observable from the sensors. The control system will need to generate an estimate of the state and act upon it.*

The first three requirements above suggest that a control system cannot explicitly encode what action to take for each possible combination of sensor readings it receives. Instead, it must have at its disposal a general method for determining the best action to take given the sensor readings. Conceptually, such a general method can be quite simple. For example, if we could directly measure the amount of air flowing into an internal combustion engine, basic chemistry would require that we provide fuel in the ratio of 14.7 to 1 to the air. Our parameter (the amount of air) can vary continuously across the real numbers, and the same control law ( $fuel = air/14.7$ ) informs us how to continuously vary the fuel in response. Unfortunately the fourth and fifth requirements make the application of traditional continuous control laws impractical. A continuous mathematical function that takes as an input the current configuration of the valves in the Cassini propulsion system and computes which valve to open or close first would be quite difficult to derive, encode and understand. A continuous mathematical function that takes as input the current sensor readings and returns the position of each valve in the Cassini system, taking into account sensor failures, redundant information from flow sensors along the same pipe, and so on, would be equally



unmanageable. What we require is a method of simply and compactly specifying a discrete controller that applies over all possible states of the system.

Because of the number of states, we cannot specify the discrete control system as a table of sensor values and the discrete decisions that must be made in response. Specifying a control system via rules that determine what action to take, such as in an expert system or in the “if then else” statements of a program, has the advantage that the rules can be fired regardless of the sensor values that are received. The disadvantage is that it can be quite difficult to determine what states a set of rules or program statements actually cover and how the addition of new rules will affect the behavior of any existing rules. In the case of Cassini, spacecraft engineers performed a large amount of analysis to determine the most likely failures of the system, how to diagnose those failures from the sensor values, and the appropriate response to each. While this provided a highly capable discrete control system, the analysis and software development required came at a cost of over a million dollars per critical segment of the mission (*e.g.*, orbital insertion) and the overall development time was several years. Thus our requirements that control systems be fast and inexpensive to develop is not met.

What makes these techniques difficult and expensive to employ on complex systems is that they are aimed at encoding the discrete decision processes that will be used to perform state identification and control. In essence, encoding the process requires the control system developer to perform the decision process by thinking through how a component failure will effect the behavior of the overall hardware system, how that failure will be diagnosable given the sensors, and what the response should be. If the control system must identify and account for failures in sensors or actuators that will change how the overall hardware system responds, performing the system level reasoning required to create the control system can be quite complex. The more components and subsystems comprising the hardware being controlled, the more complex this system-level reasoning grows, and the more expensive and less maintainable the resulting encoding becomes.

One approach to avoiding the cost of encoding a complex process is to encode it only once. Computer graphics are a good analogy. An artist using a computer graphics system

does not encode a specialized process for drawing a still life. Instead, the artist describes the local properties of the objects in the scene, such as shape, texture and position. In order to generate a photo-realistic picture of the entire scene, the computer applies standard graphics algorithms to the local descriptions of the objects. When the local properties of an object are changed, the algorithms are re-applied and a new scene is correctly generated. In essence the artist describes *what* is in the scene, while the graphics algorithms capture *how* to draw any scene that can be described in a scene description language. Similarly, our approach will be to introduce standard, reusable algorithms for discrete state identification and control that build upon techniques from the model-based diagnosis literature. To use the algorithms, the control system developer will describe, or model, the local characteristics of the components of a hardware system using a modeling language. These local models will then be combined by the algorithms to perform system level state identification and control over any state the model can attain. We will refer to the resulting system as a model-based discrete control system. The nature of the models and algorithms that make up a model-based discrete control system, and how they satisfy all of the requirements we have introduced, are the subject of the next section.

### 1.3 Model-based Discrete Control Systems

In order to develop a model-based discrete control system, we require a language for specifying a model of the components of our hardware, and a set of algorithms that make use of our models to perform control. This section provides an overview of the models and algorithms used in this work through an example.

**Example 2** Figure 1.4 represents a simple valve system that will be used as an example throughout this document. The helium tank pressurizes the system and the valves, if open, allow a gas flow. The valve driver unit (VDU) commands two valves via the data bus represented by dashed lines. The valves are commanded in parallel. The VDU can command both valves open or both valves closed.

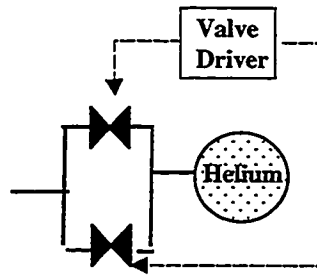


Figure 1.4: The Valve Driver Example

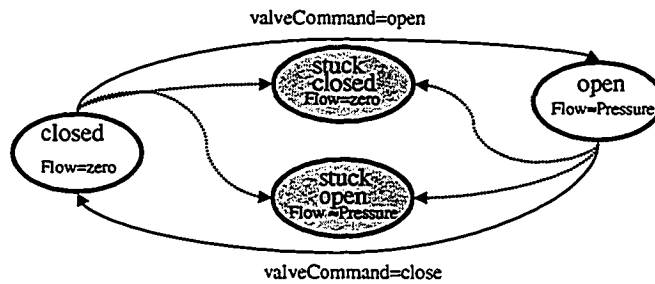


Figure 1.5: The Automaton Representing a Valve

A model of this hardware system must specify the components of the system (*e.g.*, there are two valves, a tank and a driver). For each component, the model must specify the possible states, referred to as *modes*, the component may occupy (*e.g.*, a valve may be open, closed, stuck open, or stuck closed). For each mode, the model must specify the component's behavior (*e.g.*, a closed valve prevents flow) and transitions (*e.g.*, when commanded open, a closed valve usually opens but may stick closed with some probability  $p$ ). All of this information can be encoded using an automaton to represent each component. For example, a valve might be represented as shown in Figure 1.5.

The ovals in Figure 1.5 represent the possible modes of the valve, *open*, *closed*, *stuck open* and *stuck closed*. Each mode includes a partial description of how the valve behaves in that mode. For example, when the valve is in the closed mode, the flow through the valve is zero. The arcs specify how the mode changes when an action is taken. Starting in the closed mode, when the command to open is given, the most likely outcome is that the valve moves to the open mode via the darker arc. The lighter arcs represent less likely failure transitions to the stuck open or stuck closed mode that may occur.

Using a single component, we can develop a basic intuition for how a discrete control algorithm might work. For the sake of simplicity, the algorithm is broken down into a method for estimating the state of the system given the sensors and a method for determining the best action to take given the state estimate. Suppose we know a valve to be in the closed mode, and we issue the command to open the valve. We then receive an observation of the flow and pressure, and wish to determine the new state of the valve. Suppose the flow reported by the sensor is zero and the pressure is high. We investigate each possible transition from the closed mode in turn. If the valve took the likely transition to the open mode, the flow through the valve would be proportional to the pressure. It is not, so this transition, although likely, is ruled out. Similarly, the less likely transition to the stuck open mode is ruled out by the observations. The only transition consistent with the observations is the stuck closed mode, and this becomes our state estimate. The basic intuition is that *state identification is a search over the transitions of the hardware model to find a mode that is consistent with the observations.*

Choice of a control action is accomplished in a similar manner. Suppose we again know the valve to be in the closed mode. We wish to have flow through the valve. We first check to see if the current mode allows flow. It does not, so we must find a path to a mode that does. We cannot use any of the arcs to failure modes in our path, as we cannot force failures to occur. Instead, we must use only the commandable (darker) arcs. In this case, there is only one arc from the closed mode to the open mode. Fortunately, in the open mode there must be flow and our search ends. The basic intuition is that *action selection is a search over the transitions of the hardware model to find a mode that enforces the desired conditions.* In the next two sections, we present an overview of the challenges in creating diagnosis and action selection algorithms based upon these intuitions, and the techniques we will use to address them.

#### 1.4 Overview of State Estimation

We can represent the basic features of the automaton of Figure 1.5 in an influence diagram, as shown in Figure 1.6. Each arc represents the existence of one or more constraints between two variables. The straight arcs represent that the current mode of the valve, the valve

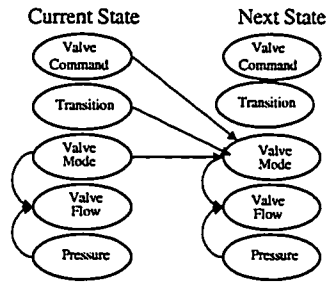


Figure 1.6: An Influence Diagram for the Valve

command and the transition taken determine the next mode of the valve. The curved arc represents that in each state the pressure and mode of the valve determine whether there is flow at the valve. As with the automaton, we can perform state identification by performing a search over the possible transitions and examining how each influences the observations in the next state. We can also develop an automaton for the VDU and helium tank and represent them as influence diagrams. A combined influence diagram for the VDU system is shown in Figure 1.7. There are variables representing the modes and transitions of the two valves, the VDU and the tank, as well as other variables. As in the valve diagram of Figure 1.6, the pressure in the system influences the flow through the valves. In addition, the mode of the helium tank (*i.e.*, OK or ruptured) determines whether there is pressure. Similarly, the mode and command input to the VDU determines the commands sent to the valves. As with the simpler model, we can perform state identification via a search on the transitions. However, there are now four transitions which require choices, leading to 64 possible combinations for this small model. This highlights that state identification can be cast as a *combinatorial optimization* problem: we are interested in the best (in this case, maximum probability) combinations of choices for the transitions that make the next state consistent with the observations. In Figure 1.8 we see the influence diagram for the VDU system as it is commanded four times. Note how the pressure at valve V1 at time step 2 is dependent upon the valve mode at time step 2. The mode is in turn dependent upon the mode at time step 1, the V1 transition at time step 1, and so on. Thus, computing a state estimate for time step 4 requires a search over all of the transitions in the model. We'll refer to a sequence of transitions as a *trajectory*. If there are 64 transition combinations per

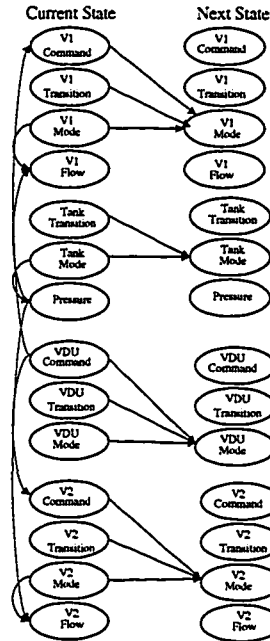


Figure 1.7: An Influence Diagram for the VDU, Two Valves and a Tank. Some arcs not shown for clarity.

time step, and we require a trajectory made up of a sequence of four trajectory combinations, then there are 16,777,216 possible trajectories. While the simple method of choosing trajectories by considering each possible trajectory and what observations it predicts is correct, it clearly cannot be used in practice because of the number of possible trajectories. We must therefore employ a technique that does not require us to consider every possible trajectory.

Given a model of a physical system and a sequence of commands and observations received over time, we can represent the problem of determining the system's state using a *hidden Markov model* or HMM, a standard representation from operations research and computer science. Given a two-step influence diagram such as Figure 1.7, standard HMM techniques can compute a state estimate after an arbitrary number of commands and observations have been received. This state estimate takes the form of a *belief state*. A belief state expresses the likelihood of each possible configuration of the system. The belief state is a sufficient statistic, in that it captures all knowledge about the current state of the system

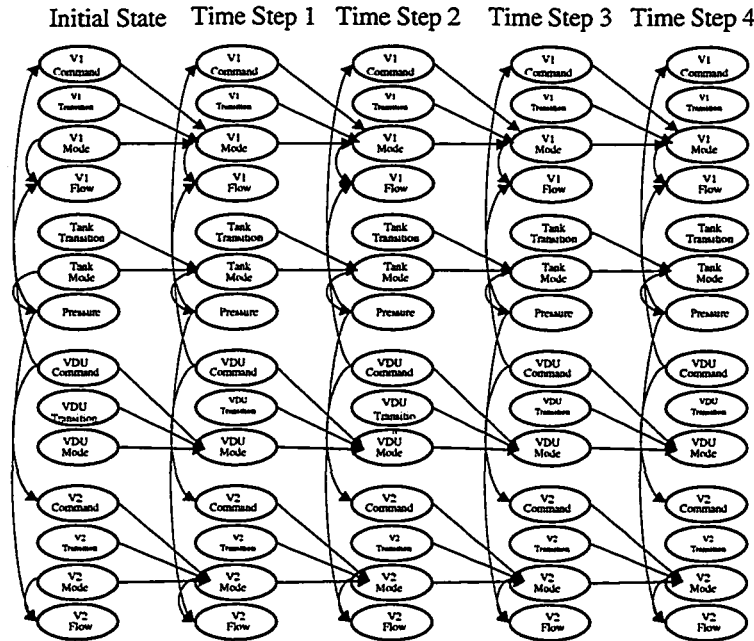


Figure 1.8: An Influence Diagram for the VDU System over 4 Time Steps

contained within the history of commands and observations. Once the belief state is computed, there is no need to retain or re-examine any previous commands or observations. Thus we do not need to enumerate every possible trajectory of the system over time. Unfortunately, computation of the belief state still requires computing the probability of every possible state of the system, which is out of the question for problems of the size we hope to address.

It is important to note that while we cannot compute the probability of every possible state of the system, we also cannot go to the other extreme and assume that we will be able to uniquely identify and track the one true state of the system given the observations we have received. Many complex physical systems are only partially observable, or observations may be costly. For example, sensors dedicated to measuring the internal state of spacecraft are usually quite minimal due to power and weight constraints. Thus, if we diagnose a fault aboard a spacecraft, it may be impossible to do better than finding a small set of failures that are equally likely given the limited set of observations. For example, consider the simplified propulsion system of figure Figure 1.9. If the engine fails to produce thrust, we

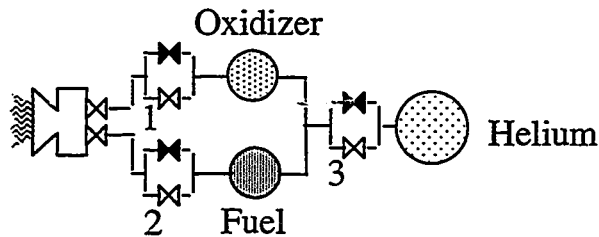


Figure 1.9: Simplified propulsion system schematic.

may not be able to immediately distinguish whether the valve at position 1, 2 or 3 is stuck closed. We must therefore be able to track multiple trajectories of the system.

Rather than computing a complete belief state that accounts for every trajectory after each command, as we would do with a traditional HMM technique, we will focus on maintaining the most likely trajectories of the system given the commands and observations received thus far. Each of these trajectories ends in a current state of the system that we consider to be among the most likely. Unfortunately, subsequent observations may provide additional information that forces us to reduce the likelihood or rule out some of states of the system we previously considered likely. The new most likely trajectory is then one we have not yet considered, and finding it will require re-examination of the history of the system. Thus we are back to performing trajectory identification on growing structures similar to Figure 1.8. The naive approach of simply growing the model each time the system is commanded and performing a complete search over the trajectories is clearly insufficient. We will first consider modifying the search algorithm for state identification. Using algorithms inspired by work in the field of model-based diagnosis, we will in the average case vastly speed up the search process. One such technique is to implicitly rule out large numbers of trajectories that are inconsistent with the observations without ever explicitly considering them. We will then limit the size of the model that is to be searched through techniques such as not explicitly representing every variable at every time step. These techniques together comprise a practical state estimation algorithm that has been demonstrated on complex models.

In summary, our state estimation technique will track the trajectories of the system that appear most likely given the observations received thus far, and in doing so yield a set of



likely current states. As we receive additional observations, it may be the case that the trajectories we have chosen to track were very unlikely to have produced the new observations, or could not have possibly produced the new observations at all. We will then use history information to generate additional trajectories we previously considered unlikely in order to replenish our set of likely current states.

## **1.5 Overview of Planning**

The preceding section gave provided an overview of how we will compute a set of likely current states of a physical apparatus, and how we will maintain that set over time. Given that the control system no longer knows with certainty what state the apparatus occupies, the question arises of how should one choose actions. We would like to choose actions that move the apparatus from its current state to a state that has some desirable property, referred to as a goal state. Our problem is that we need to generate a sequence of actions, or a plan, that achieves our goals even though the exact failure that occurred, and thus the exact state of the apparatus in which our plan will execute, cannot be determined. One simple approach is to plan as if the apparatus were in one of the states, and hope that the observations received during execution of the plan will reveal which state is the actual state. For example, again consider the simplified propulsion system of Figure 1.9 and imagine that we have diagnosed that one of the valves at position 1, 2 or 3 is stuck closed. We might assume that it is valve 1, the oxidizer valve, that is stuck, and choose actions that open the backup valve adjacent to valve 1.

Unfortunately, an action that achieves some desirable goal when the apparatus is in one possible state may be ineffective or precipitate a disaster if the apparatus is in fact in another state. For example, it is equally likely that valve 3 and not valve 1 is stuck closed, and therefore the oxidizer tank is not being pressurized by the helium tank. Consider if it is not safe to adjust the valving on the oxidizer tank when it is not pressurized, perhaps because of reversion. In this case, our actions chosen only to operate correctly in a single state would be dangerous if another equally likely state were actually in force. Thus when choosing actions we must in some manner consider each possible state that the apparatus may occupy when we begin executing actions.

One formulation of planning that suits our application is the *conformant planning problem*. Intuitively, conformant planning is the problem of generating a plan that moves a system from any one of a number of possible initial states to a state that satisfies a set of goal predicates. The challenge of conformant planning lies in the fact that the effects of a plan when executed in one state may be different and highly undesirable when the plan is executed in a different state. Thus one cannot choose an action based on its desired effect given one possible initial state of the system (called a *world* in the conformant planning literature) without in some way considering its unintended effects when it is executed in all other possible initial states. The traditional approach to conformant planning has been to consider the effects of each action under consideration across all worlds simultaneously. This of course has a large impact on the computation and space required to generate a plan. In this work, we take a unique approach to conformant planning in that we attempt find a plan that works in a single world and extend it to work in all worlds. To do this, we plan in a single world at a time using a deterministic planner. We use the plan generated in each world to influence how plans are generated in the remaining worlds, in order guide the planner toward producing a plan that works in all worlds. In effect, we attempt to divide a complex, conformant planning problem into a set of simpler, single-world planning problems. This conformant planning technique, referred to as *fragment-based conformant planning problem*. We have implemented several variations of fragment-based conformant planners, and have tested them on conformant planning problems from domain of spacecraft reconfiguration and from the planning literature.

A conformant plan is a plan that achieves *all* of our goals from *any* of the possible initial states of our apparatus. Our experience suggests that conformant planning is too restrictive for the type of robust, autonomous operations we seek in spacecraft and other real-world systems. Simply put, it may be the case that no conformant plan can be found, because one or more of the possible initial states of the system contain severe failures and no conformant plan exists, or because we cannot find a conformant plan by the time we need to begin executing actions. In this case, we will not be able to generate a plan that achieves all goals from any initial state. It may however be possible to find plans that achieve some of

the goals from all initial states, all goals from some initial states, or different sets of goals in different sets of initial states, each of which has a different utility to the user. Since we are interested in autonomous systems, we require a way to automatically reduce the scope of the planning problem when a conformant plan that achieves all goals in all worlds cannot be found in the available time. This will require a strategy for choosing the goals and worlds for which a plan will be attempted when a plan cannot be found for all. For many applications, we would like the strategy to ensure that some valid plan of action is available when we must act. The strategy for must also take into consideration the utility or relative importance the user assigns to various combinations of goals to be achieved and initial states to be handled.

We have developed a planning algorithm, the Safe, Conformant, Optimizing Planning Engine, or *SCOPE*, that has these properties. *SCOPE* attempts to find a conformant plan for a subset of the complete sets of initial states and goals of its choosing. It chooses states and goals so as to attempt to respect the user's utility assignments, and can employ various strategies for ensuring a plan exists whenever planning is interrupted (*anytime planning*) or exploring which worlds or goals are making a problem difficult to solve (*learning*). Our fragment-based conformant planner, which is unique in its ability to incrementally expand the set of worlds upon which its plan is conformant, is used as a subroutine to the *SCOPE* planner. The *SCOPE* planner has been implemented, and a variety of strategies for choosing initial states and goals have been explored.

## 1.6 A Model-based Health Maintenance Capability

By combining a state estimation capability with planning, we can form a model-based capability for system health maintenance or control, similar to those provided by Livingstone (Williams & Nayak 1996) and demonstrated on a spacecraft as a part of the Remote Agent system (Bernard *et al.* 1998). In such a system, a model of the apparatus being controlled is combined with observations received from the system by a diagnosis or state estimation algorithm to determine the current state of the apparatus. The planner then finds a sequence of actions that move the apparatus from its current state to a goal state. As the actions are executed, the new current state is determined from the resulting observations,

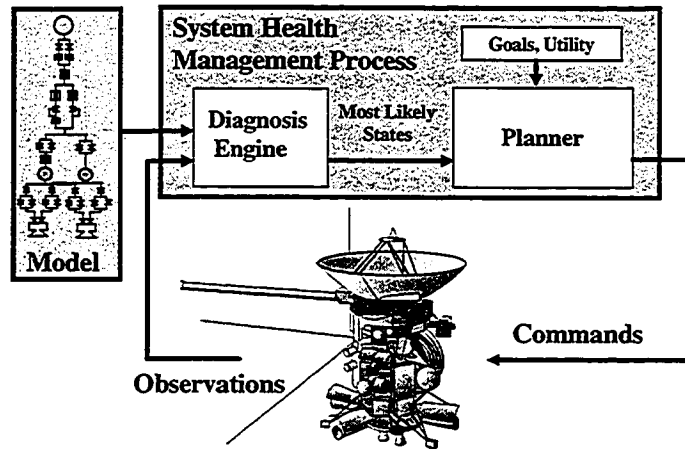


Figure 1.10: Model-based Health Maintenance

and the process repeats. The spacecraft is considered reliable enough that when finding a reconfiguration to mitigate the initial failure, we do not consider the effect of possible additional failures during the reconfiguration. If reconfiguration fails, we simply diagnose the additional failure and find a new reconfiguration. A model-based health maintenance system is illustrated schematically in Figure 1.10.

A control system making use of our diagnosis and planning capabilities builds on systems such as Livingstone primarily in two ways. First, it handles uncertainty in a more explicit manner. Our diagnosis work explicitly tracks the multiple possible current states caused by partial observability, and the planning work attempts to find actions that take all of these states into account. Livingstone and similar systems in model-based diagnosis and robotic control commit to one of the possible states, and plan as if that state were the true state. Second, our planner is able to balance the conflicting concerns of goal achievement, uncertainty and resource management. If a plan for all goals and possible states cannot be found within time and memory bounds, the planner discards goals or possible states in an order specified by the system designer. Livingstone's recovery capability and many other planners return no plan if the full set of goals cannot be satisfied within the time and space allotted for computation.

In addition to being more robust to uncertainty and resource constraints, we have the

opportunity to investigate alternative ways of integrating the diagnosis and planning components of this type of architecture. For example, a plan will be generated that applies in potentially many initial states. An early action in the plan might result in observations that rule out many of the states for which the plan applies. While this does not prevent the plan from reaching its goals in the remaining states, simpler plans or plans that reach additional goals from the reduced set of states may exist. Thus while the initial portion of the original plan is still executing, we may wish to regenerate the remainder of the plan without the worlds that have been ruled out by the execution thus far.

## 1.7 Document Overview

In the following chapters we first address state identification. We begin with a background in state identification from the areas of partially observable Markov decision processes and model-based diagnosis. We then relate state identification to trajectory identification. We introduce the trajectory system representation that implements, in propositional logic, the intuitions of the influence diagrams presented above. Then follows a discussion of several search algorithms for the trajectory identification within the transition system representation. Turning to the issue of model size, we introduce optimizations and approximations that prevent the transition system representation from growing unboundedly as time passes. These modifications maintain the ability of a trajectory search algorithm to revise its assessment of how the system evolved in the past in order to reconsider trajectories it had previously dismissed as unlikely. Finally on the subject of state identification we present the results of testing a system that embodies these ideas, *L2* (for *Livingstone2*), on scenarios developed while applying *Livingstone* and *L2* within NASA.

On the subject of action selection, we first introduce our novel approach to conformant planning, fragment-based conformant planning, followed by experimental results comparing our fragment-based conformant planner against the best conformant planners currently available. We then motivate the expansion of our planning algorithm from a conformant planner to an optimizing planner, SCOPE, that selects a subset of goals and initial states for which it will attempt to find a conformant plan. We then present results for SCOPE and a survey of related work. Finally, we conclude with an analysis of how this work has

impacted the state of the art, which types of problems are best suited for solution by this approach, and areas for future work.

## Chapter 2

### Related Work in State Identification

In this chapter, we begin discussion of the state estimation portion of a discrete controller. Intuitively, we will be commanding a system that is not completely reliable and will be receiving observations in response. The task of state estimation is to determine the likelihood of each possible state of the system based upon the commands and observations received thus far. Based upon these likelihoods, the appropriate next action may then be selected. We begin the discussion with basic definitions.

**Definition 1** A *belief state* is a probability distribution over the possible states of the system. The likelihood assigned to a state by the belief state represents the controller's belief that the system is currently occupying that state. If  $s$  is a system state, we will write  $b(s)$  for the probability value assigned to state  $s$  by belief state  $b$ .

**Definition 2** The task of state estimation is as follows. Given a model of a system, a sequence of non-deterministic actions taken by the system, and a sequence of observations, compute the belief state.

The state estimation techniques developed in this document draw upon the techniques from two existing formulations of the discrete control problem. The first formulation is the partially observable Markov decision process. The second formulation, model-based diagnosis, is a related but independently developed formulation of the problem that brings with it a powerful set of algorithms. The two formulations, the basic techniques associated with them, and their shortcomings with respect to our domains of interest are discussed below.

## 2.1 Partially Observable Markov Decision Processes

A commonly used formalization of the discrete control problem is as a Markov decision process (Sondik 1971). For a process to be Markov, the current state and action must provide all of the information available for predicting the next state. That is, if we know the current state of the system, knowing the previous state of the system cannot add information when attempting to predict the next state of the system. We define a Markov decision process as follows.

**Definition 3** A *Markov decision process* is defined by the tuple  $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ , where

- $\mathcal{S}$  is the finite set of states of the system being tracked
- $\mathcal{A}$  is a finite set of actions
- $\mathcal{T}$  is a state transition model of the environment, which is a function mapping elements of  $\mathcal{S} \times \mathcal{A}$  into discrete probability distributions over  $\mathcal{S}$ . The actions are non-deterministic, so we write  $T(s, a, s')$  for the probability that the environment will make a transition from state  $s$  to state  $s'$  when action  $a$  is taken.
- $\mathcal{R}$  is a reward function mapping  $\mathcal{S}$  to  $\mathbb{R}$  that specifies the instantaneous reward that the agent derives from entering state  $s$ . The reward is used in action selection, and is not discussed further in this chapter.

In a Markov decision process, the state of the system is assumed to be directly observable. The probability that an action executed in the current state  $s$  will result in a new state  $s'$  is determined by  $T(s, a, s')$ . Once the action is taken, the resulting state is directly observed. Hence there is no state estimation problem. When the state is not completely observable, we must add a model of observations, to create a partially observable Markov model.

**Definition 4** A *partially observable Markov decision process* is defined by the tuple  $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{O}, \mathcal{T}, \mathcal{R} \rangle$ , where

- $\mathcal{O}$  is a finite set of possible observations
- $\mathcal{O}$  is an observation function, mapping  $\mathcal{S}$  into discrete probability distributions over  $\mathcal{O}$ . We write  $O(s, o)$  for the probability of making observation  $o$  from state  $s$ .



Though the current state is not known with certainty in a POMDP, the Markov assumption that knowledge about previous states will not improve our prediction of the next state will still prove useful in designing a state estimator. Such an estimator can be constructed out of  $T$  and  $O$  by straightforward application of Bayes' rule. Given a belief state  $b$ , the output of the state estimator is an updated belief state,  $b'$ . For each state  $s'$ ,  $b'(s')$  can be determined from the previous belief state  $b$ , the previous action  $a$ , and the current observation  $o$ . We will compute  $b'(s')$  in two steps. Given our current belief state, we first can compute our new belief the system is in state  $s'$  after executing action  $a$ , but prior to receiving any observations, denoted  $p(s')$ ,

$$p(s') = \sum_{s \in \mathcal{S}} T(s, a, s') b(s). \quad (2.1)$$

This equation is a simple consequence of the Markov property. Intuitively, every state  $s$  we could have been in has some likelihood of depositing us into  $s'$  given the action  $a$ . Each  $s$  contributes to  $s'$  according to the likelihood we were in state  $s$  and the likelihood that that state  $s$  transitioned to  $s'$ . Once  $p$  is computed, we find the new belief in  $s'$  conditioned upon the observation  $o$  we have received.

$$b'(s') = \frac{O(s', o) p(s')}{\Pr(o | a, b)} \quad (2.2)$$

This is simply Bayes' rule. Intuitively, conditioning on the observation using Bayes' rule redistributes the probability mass according to how much more likely or unlikely it was to see the observed observation  $o$  in state  $s'$  than in general.  $\Pr(o | a, b)$  is simply a normalizing factor that represents the likelihood of seeing  $o$  at all given our previous belief state. Specifically,  $\Pr(o | a, b)$  is the marginalized likelihood of seeing  $o$  given action  $a$  and our previous belief state  $b$ , defined as

$$\Pr(o | a, b) = \sum_{s' \in \mathcal{S}} O(a, s', o) \sum_{s \in \mathcal{S}} T(s, a, s') b(s). \quad (2.3)$$

The resulting  $b(s')$  function ensures that the current belief state accurately summarizes all available knowledge. That is, by repeatedly applying Equation 2.2, we maintain a belief state that captures all information contained in an arbitrarily long stream of actions and observations. Thus we have a very simple and elegant solution to discrete state estimation.

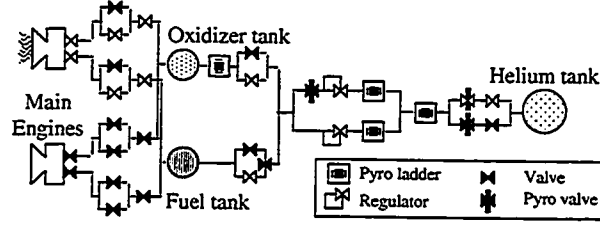


Figure 2.1: Propulsion system schematic.

Unfortunately, it is not practical to directly apply this state estimator in the domains we seek to address.

Consider the problem of determining the likelihood of the possible states of the propulsion subsystem of Figure 2.1. Computing a belief state via Equation 2.2 requires enumeration of the state space. That is, to compute  $b(s')$  we must consider  $b(s)$  for every  $s \in \mathcal{S}$ . The propulsion subsystem has 38 components with an average of 3 states each. The size of  $\mathcal{S}$  is approximately  $10^{18}$ . More complete spacecraft models capture 150 or more components averaging 4 states, yielding a state space of  $2^{300}$  or more<sup>1</sup> and making exact computation of  $b(s')$  implausible.

One alternative is to track an approximation whose computation does not require enumeration of the state space. Boyen and Koller (Boyen & Koller 1998), for example, provide an approximate, factored belief state with a bounded error that can be updated without enumerating the state space. Intuitively, the error bound relies upon the stochasticity of the underlying system, parameterized by the problem's *mixing rate*, to continually smear both the approximate and true distributions, exponentially reducing rather than compounding errors over time. Unfortunately, the systems we consider have inadequate mixing rates. Intuitively, when monitoring the internal state of a complex device such as a spacecraft, the device may behave as if it were deterministic for long periods, then exhibit a failure, then return to apparent determinism. There is no process in place with sufficient stochasticity to quickly contract an arbitrary error introduced by a factored approximation. A particle filter (Isard & Blake 1998; Doucet 1998) is another approximate representation for a belief

<sup>1</sup>Or

2,037,035,976,334,486,086,268,445,688,409,378,161,051,468,393,665,936,250,636,140,449,354,381,299,763,336,-706,183,397,376

state that does not require enumerating the entire state space. A particle filter approximates the belief state via sampling. Initially, a subset of the states of the statespace are selected for representation. This selection may be uniform over the statespace, or represent some knowledge about the initial state of the system being tracked. Conceptually, there is a particle at that point in the space. When an action is taken and an observation received, the likelihood of the state each particle represents is updated via Equations 2.1 and 2.2. The state space is then re-sampled, using the updated likelihoods of the particles as a bias. States near likely particles are more likely to be sampled, while particles that represent states that have become unlikely may be eliminated. Conceptually, the particles condense toward regions of the state space that predict the current behavior of the system. Unfortunately, a particle filter is not well-suited to tracking the kind of abrupt, discontinuous failures we seek to diagnose. Intuitively, the re-sampling step of the algorithm is unlikely to place particles into failure states, since the likelihood of entering a failure state is low. Thus when a failure occurs, it is unlikely that the particle filter will have a particle representing that state and its likelihood. Dearden and Clancy (Dearden & Clancy 2002) and Thrun et al. (Thrun, Langford, & Verma 2002) suggest methods for modifying the re-sampling step of the particle filter algorithm in an attempt to address this shortcoming.

## **2.2 Model-Based Diagnosis**

Techniques from model-based diagnosis take a different approach, incrementally generating members of the belief state in most-likely first order (de Kleer & Williams 1987; 1989). In this approach, the device is typically modeled as a set of components. Each component has a set of variables and one or more states, or modes, that it can occupy. Each mode has a (typically) propositional model that constrains the values of the components variables. Thus, setting the mode of each component induces a set of constraints on the variables of the complete model. Some of these variables are directly observable from the device, meaning that certain assignments of the modes will not be consistent with the observations. The task is then to assign each component's mode so as to cause consistency with the observations.

Component modes that represent failures are assigned a cost corresponding to the prior

probability of that failure occurring in that component. An assumption is generally made that failures of components occur independently. Thus the probability of a set of mode assignments is the product of the probability of the mode assignments. Thus starting with the lowest cost assignment (each device in its nominal mode) we can consider all complete mode assignments in order of total likelihood until an assignment consistent with the observations is found. This mode assignment represents the most likely state of the system. Using this simple best-first procedure, many inconsistent mode assignments may be found before a consistent assignment is found. Note however that if a partial assignment to the modes introduces a set of constraints that causes an inconsistency, every full assignment that contains this partial assignment is also inconsistent. This partial assignment to the modes and observations is called a conflict. As candidate mode assignments are ruled out because they are inconsistent, conflicts are found and recorded. No assignment that contains a known conflict will ever be brought up for consideration. This has the effect of potentially ruling out large portions of the statespace from consideration, and can dramatically focus the search for a consistent assignment. Conflict-directed best first search, *CBFS*, performs best-first search on those parts of the search space not yet known to contain a conflict. Sherlock (de Kleer & Williams 1989) introduces *CBFS* and the application of diagnosing the failure modes within a stateless system such as combinatorial circuits. Conceptually, this type of system tracks for a single step a limited HMM where there is a single action, specified compositionally across the modes, that either has no effect or causes some number of components to fail. The belief state for the HMM after this one action is executed is incrementally generated until the desired number of possible states or amount of probability mass is accumulated.

*Livingstone* (Williams & Nayak 1996) adds to Sherlock the ability to transition a component's modes between nominal modes in response to an action in addition to moving from a nominal mode to a failure. After each action is performed, *Livingstone* uses *CBFS* to enumerate a small number of most likely mode assignments given the current observations and the previous mode assignments. Each partial belief state is made up of only descendants of the previous  $n$  most likely states, which were determined using only previous observations.

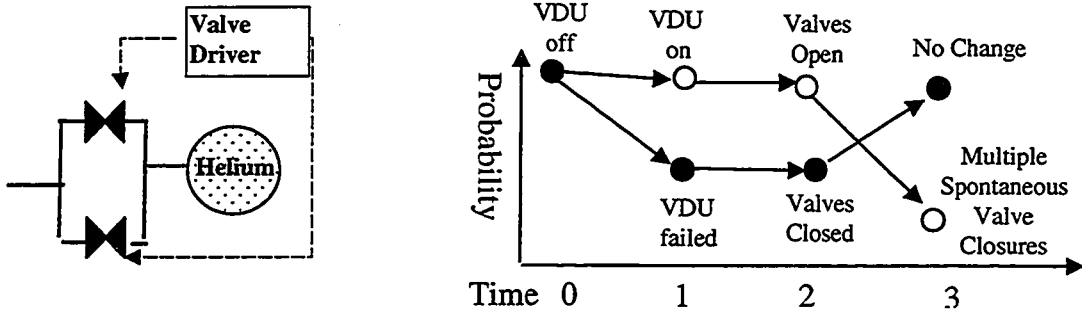


Figure 2.2: Evolution of a Valve Driver Unit and Valves

*Livingstone* tracks the  $n$  approximately most likely states of the system. This approximation is extremely efficient and well suited to the problem of tracking the internal state of a machine, where the likelihood of the nominal or expected transition dominates, and immediate observations often rule out the nominal trajectory when a failure occurs. The task then becomes one of diagnosing the most likely system transition, chosen from combinations of component transitions, that would be consistent with the unexpected observations. Using this technique, *Livingstone* is able to perform approximate state identification and reconfiguration of systems with hundreds of state variables. It has been applied to the control of a number of systems within NASA and is an integral part of the Remote Agent architecture demonstrated in-flight on the Deep Space 1 spacecraft in 1999 (Muscettola *et al.* 1998; Bernard *et al.* 1998). Unfortunately, the true trajectory may not be among the most likely given only the current observations. Consider the following example.

**Example 3** Figure 2.2 reintroduces the valve system from Figure 1.4. Recall that the helium tank pressurizes the system and the VDU commands the valves to open or close in parallel. The graph to the right represents the probability of two possible trajectories. The filled circles represent the true state of the system. At time 0 the VDU is off, the valves are closed and pressure is observed at the outlet of the helium tank. At time 0 the VDU is commanded on. For the sake of illustration, consider an approximate belief state of size 1. The state wherein the VDU is on is placed into the belief state. The true state wherein the VDU is failed is discarded. At time 1, the VDU is commanded to open its valves. Since the only state in the belief state assumes the VDU is on, the single state in the updated

belief state has the VDU on and all valves open. In the true, untracked state the valves are closed, as they never received a command. After commanding the valves to open, no flow is observed downstream of the valves. Failure of the helium tank has zero probability, given the observations. Failure of the VDU in the current time step has no effect on the valves. Thus, the most likely next state consistent with the observations requires that all valves spontaneously and independently shut. Regardless of the number of valves and the unlikeliness of spontaneous closure, this transition must be taken if it exists. If it does not exist, the belief state approximation becomes empty.

While only one trajectory is tracked in this example, adding a fixed number of trajectories will not help in the general case. For any fraction of the trajectories that are tracked, an example can be constructed wherein the actual state of the system falls outside the tracked fraction. In this case, the error in the approximate list of most likely states may become arbitrarily large. The true state of the system will not be in this list of “most likely” states, and the states that are maintained on the list need not be very likely at all. Intuitively, as the true state of the system evolves and produces observations, the incorrectly tracked subset of states may need to undergo arbitrarily unlikely transitions in order to remain consistent with the observations. This mis-diagnosis can cause a controller that is relying upon this state estimate to take incorrect and potentially damaging actions. In the next chapter, we therefore propose an alternative to committing to a subset of the current belief state or maintaining an approximation of the entire belief state.

## Chapter 3

# Trajectory Identification

### 3.1 Introduction

In this chapter, we propose to maintain the information necessary to begin incrementally generating the current belief state in best-first order at any point in time. Since we do not update the entire belief state, we do not have a sufficient statistic, so a history must be maintained. We introduce a variable to represent every state variable, command and observation at every point in time and an algorithm for incrementally generating the exact belief state at any point. Duplicating the entire set of variables at each point in the history seems impractical except for short duration tasks. We apply two approximations motivated by our experience modeling physical systems for *Livingstone*. The first duplicates only a small number of carefully selected variables at each time point. This approximation is conservative in that it does not eliminate any feasible trajectories but may admit certain infeasible trajectories. These may be eliminated by future observations. The second limits the length of the history that is maintained by absorbing older variables into a single variable that grossly approximates them. This allows an approximate belief state to be generated at any point in time from a constant number of variables. The variables represent an exact model of system evolution over the recent past, an approximate model over the intermediate past, and a gross summarization over the more distant past. This allows assignment of the most likely past transitions to be revisited as new observations become available. The fewest variables, and thus the least flexibility, are allocated to segments of the system trajectory that have remained consistent with the system's observed evolution for the longest time.

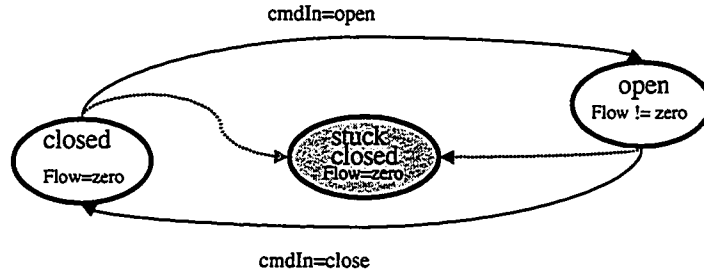


Figure 3.1: A Simplified Valve Automaton

We wish to represent the possible histories of a system composed of non-deterministic, concurrent automata given the commands issued to the automata and their output. Figure 3.1 is a slightly simplified version of the valve from Chapter 1. One failure state has been removed solely for the purpose of clarity. From these automata, we would like to create a structure for representing all possible evolutions of the valve over time. We would also like a propositional encoding so we may take advantage of techniques and intuitions developed in the model-based diagnosis world. Based on these desires, in this chapter we introduce a propositional representation of non-deterministic automata that is an extension of the formalism used in *Livingstone*. We then frame the trajectory identification problem.

Before precisely defining the representation, we will develop an intuition using Figure 3.1. Representing the behavior of the automaton within each state is straightforward. Let *Valve* be a variable representing the possible states of the automaton. The domain of *Valve* is  $\{open, closed, stuck\}$ . Let *Flow* be a variable representing the flow through the valve, of domain  $\{zero, non.zero\}$ . A propositional model of the open state of the automaton is then simply:

$$Valve = closed \implies Flow = zero$$

The constraints within each of the other states of the automaton can be similarly captured. For our future convenience, we will refer to the set of formulas introduced to model the behavior of all of the states of all automata in the system as  $\mathcal{M}_\Sigma$ . Capturing the transitions of the valve automaton in propositional logic is slightly more challenging. There is no operator to capture that when a command is given to the valve, it non-deterministically chooses with some probability to transition from open to closed or open to stuck closed. A



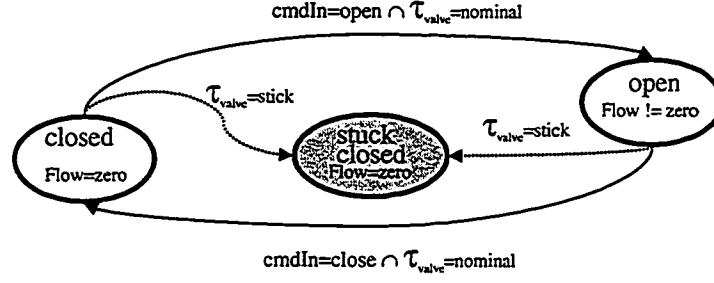


Figure 3.2: Simplified Valve Automaton with  $\tau$  Variables

simple way to capture this non-determinism in the valve is to introduce a choice variable  $\tau_{valve}$ . Figure 3.2 illustrates the augmented automaton. We may now simply model the choice of transitions taken from the closed state, for example, as the choice of assignments to the free variable  $\tau_{valve}$ . To represent the possible outcomes of the open command at time  $t$ , we must introduce variables to represent the valve at times  $t$  and  $t + 1$ , and a variable to represent the non-deterministic choice  $\tau_{valve}$  at time  $t$ . The transitions from the closed state of the valve automaton can then be modeled by the following formulas:

$$\begin{aligned}
 Valve_t = closed \wedge \tau_{valve,t} = nominal &\implies Valve_{t+1} = open \\
 \tau_{valve,t} = stick &\implies Valve_{t+1} = stuck
 \end{aligned}$$

For our future convenience, we will refer to the set of formulas introduced to model the behavior the transitions of each automaton in the system as  $\mathcal{M}_{\mathcal{T}}$ . A trajectory of the valve is an assignment to the variables  $\tau_{valve,0}, \tau_{valve,1}, \tau_{valve,2}, \dots$ . The prior probability of the valve sticking at any point can be captured as the probability that Nature makes the assignment  $\tau_{valve,t} = stick$ . Given the appropriate independence assumptions, the set of valve trajectories can be incrementally enumerated in order of prior probability. Trajectories that are inconsistent with  $\mathcal{M}_{\Sigma}$  given the observations such as the actual flow observed, need not be considered. This is the kernel of the our approach to state estimation problem. The remainder of this chapter introduces the transition system formalism more precisely and elaborates on the assumptions being made.

### 3.2 Transition systems

**Definition 5** A transition system  $\mathbf{T}$  is a tuple  $\langle \Sigma, \mathcal{T}, \mathcal{D}, \mathcal{C}, \mathcal{M}_{\Sigma}, \mathcal{M}_{\mathcal{T}}, \Gamma \rangle$ , where

- $\Sigma$  is a set of *state variables* representing the state of each automaton. Let  $n$  denote the number of automata and  $m$  denote the number of discrete, synchronous time steps over which the state is to be tracked.  $\Sigma$  then contains  $m \times n$  variables.  $\Sigma_t$  will denote the set of state variables representing the state of the system at time step  $t$ . Each state variable  $y$  ranges over a finite domain denoted  $\delta(y)$ . The temporal variable representing the occurrence of variable  $y$  at time step  $t$  is denoted  $y_t$ .
- $\mathcal{T}$  is a set of *transition variables*. The transition variable that represents the non-determinism in the transition of state variable  $y$  from time  $t$  to  $t + 1$  is denoted  $\tau_{y,t}$ . That is, if there are  $n$  non-deterministic outcomes of the transition in the value of  $y$ ,  $\tau_{y,t}$  will have a domain of size  $n$ .
- $\Gamma$  represents a likelihood function on  $\mathcal{T}$ . The exact nature of  $\Gamma$  is discussed below. Conceptually  $\Gamma(\tau_{y,t})$  represents the probability distribution over the outcomes of the transitions of variable  $y$ .
- $\mathcal{D}$  is a finite set of *dependent variables*.
- $\mathcal{C}$  is a finite set of *command variables*.
- State  $s_t$  is an assignment to  $\Sigma_t \cup \mathcal{T}_t \cup \mathcal{D}_t \cup \mathcal{C}_t$
- $\mathcal{M}_\Sigma$  is a propositional formula over  $\Sigma_t$  and  $\mathcal{D}_t$  that specifies the feasible subset of the state space. A state is feasible if it makes an assignment to  $\Sigma_t \cup \mathcal{D}_t$  that is consistent with  $\mathcal{M}_\Sigma$ .
- $\mathcal{M}_\mathcal{T}$  is a propositional formula over  $\Sigma_t, \mathcal{D}_t, \mathcal{C}_t, \mathcal{T}_t$  and  $\Sigma_{t+1}$  that specifies the feasible sequences of states.  $\mathcal{M}_\mathcal{T}$  is a conjunction of transition formulas modeling possible evolutions of  $y_t$  to  $y_{t+1}$  of the form

$$\phi_t \wedge (\tau_{y,t} = \tau^*) \implies y_{t+1} = y^*$$

where  $\phi_t$  is a propositional formula over  $\Sigma_t \cup \mathcal{D}_t \cup \mathcal{C}_t$ , and  $\tau^*$ , representing a choice among the non-deterministic transitions of  $y$ , is in  $\delta(\tau_{y,t})$ . The sequence  $s_i, s_{i+1}$  is feasible if the assignment made by  $s_i \cup s_{i+1}$  is consistent with  $\mathcal{M}_\mathcal{T}$ .

**Example 4** We introduce a transition system to model a VDU and two valves. For the sake of brevity we have omitted the helium tank. The variables corresponding to the VDU

consist of a state variable  $vdu$  representing the possible VDU states (*on*, *off*, or *failed*), the transition variable  $\tau_{vdu}$ , a command variable  $cmdin$  representing commands to the VDU or its associated valves (*on*, *off*, *open*, *close*, *none*), and a dependent variable  $cmdout$  representing the command the VDU passes on to its valves (*open*, *close*, or *none*). The feasible states of the VDU are specified by the formulas below that belong to  $\mathcal{M}_\Sigma$ :

$$\begin{aligned}
vdu = off &\implies cmdout = none \\
vdu = failed &\implies cmdout = none \\
vdu = on &\implies cmdin = open \implies cmdout = open && \wedge \\
&cmdin = close \implies cmdout = close && \wedge \\
&cmdin \neq open \wedge cmdin \neq close \implies cmdout = none
\end{aligned}$$

together with formulas like  $(vdu = on) \vee (vdu = off) \vee (vdu = failed)$  and  $(vdu \neq on) \vee (vdu \neq off) \dots$  that assert that variables have unique values. The time step subscript is omitted, indicating that all formulas refer to variables within the same time step.  $\mathcal{M}_T$  for  $\tau_{vdu}$  is as follows, where *nom* is an abbreviation for *nominal*:

$$\begin{aligned}
\tau_{vdu,t} = fail &\implies vdu_{t+1} = failed \\
\tau_{vdu,t} = nom &\implies vdu_t = off \wedge cmdin_t = on \implies vdu_{t+1} = on && \wedge \\
&vdu_t = off \wedge cmdin_t \neq on \implies vdu_{t+1} = off && \wedge \\
&vdu_t = on \wedge cmdin_t = off \implies vdu_{t+1} = off && \wedge \\
&vdu_t = on \wedge cmdin_t \neq off \implies vdu_{t+1} = on && \wedge \\
&vdu_t = failed \implies vdu_{t+1} = failed
\end{aligned}$$

The valves  $v1$  and  $v2$  each have a state variable of domain (*open*, *closed*, or *stuck*), a transition variable  $\tau_{vi}$  and a dependent variable  $flow_{vi}$  of domain (*zero*, *nonzero*). The feasible states of  $v1$  are specified by the formula below. The feasible states of  $v2$  are specified similarly.

$$\begin{aligned}
v1 = open &\implies flow_{v1} = nonzero \\
v1 = closed &\implies flow_{v1} = zero \\
v1 = stuck &\implies flow_{v1} = zero
\end{aligned}$$

$\mathcal{M}_{\mathcal{T}}$  for  $\tau_{v1}$  is shown below.  $\tau_{v2}$  has the same constraints as  $\tau_{v1}$ .

$$\begin{aligned}
\tau_{v1,t} = \text{stick} &\implies v1_{t+1} = \text{stuck} \\
\tau_{v1,t} = \text{nom} &\implies v1_t = \text{closed} \wedge \text{cmdout}_t = \text{open} \implies v1_{t+1} = \text{open} \quad \wedge \\
&\quad v1_t = \text{closed} \wedge \text{cmdout}_t \neq \text{open} \implies v1_{t+1} = \text{closed} \quad \wedge \\
&\quad v1_t = \text{open} \wedge \text{cmdout}_t = \text{closed} \implies v1_{t+1} = \text{closed} \quad \wedge \\
&\quad v1_t = \text{open} \wedge \text{cmdout}_t \neq \text{close} \implies v1_{t+1} = \text{open} \quad \wedge \\
v1_t = \text{stuck} &\implies v1_{t+1} = \text{stuck}
\end{aligned}$$

### 3.3 Trajectory Identification

**Definition 6** A *trajectory* for  $\mathbf{T}$  is a sequence of states  $s_0, s_1, \dots, s_m$  such that for all  $t$ ,  $0 < t < m$ ,  $s_t$  is consistent with  $\mathcal{M}_{\Sigma}$  and for all  $t$ ,  $0 < t < (m-1)$ ,  $s_t \cup s_{t+1}$  is consistent with  $\mathcal{M}_{\mathcal{T}}$ .

Consider the problem of determining the state of a physical process modeled by a transition system  $\mathbf{T}$  at each point in a trajectory  $s_0 \dots s_m$ . The subset of the dependent variables  $\mathcal{D}$  whose assignment corresponds to a measurement from the process will be referred to as the observations,  $\mathcal{O}$ . We are given an assignment for the initial state,  $\Sigma_0$ . In addition we are given assignments to commands  $\mathcal{C}_t$  and observations  $\mathcal{O}_t$  for all  $0 < t < m$ . The task is to choose assignments to  $\tau_{y,t}$  for all  $y$  and  $t$  so as to ensure consistency with  $\mathcal{M}_{\Sigma}$  and  $\mathcal{M}_{\mathcal{T}}$  and maximize the likelihood of the trajectory. That is to say, given a starting state, a set of commands and a set of observations, we must find the most likely sequence of transitions such that each state is consistent with the state model  $\mathcal{M}_{\Sigma}$  and the transitions are consistent with the transition model  $\mathcal{M}_{\mathcal{T}}$ . We define trajectory likelihood to be

$$\sum_{t=0}^m \sum_{y=1}^n \Gamma(\tau_{y,t})$$

This definition makes the assumption that the likelihood of the assignment to each transition variable is independent of all others. That is,  $\tau_{y,t}$  is independent of  $\tau_{x,t}$ ,  $\tau_{y,t+i}$  and  $\tau_{y,t-i}$ . This is a common assumption and has been an adequate approximation in practice. Note that this assumption does not effect the handling of single failures that manifest themselves at multiple points throughout the system (*e.g.*, a power failure causing all lights to go out).

### 3.4 Infinitesimals

In order to complete the transition system model shown in Example 4, we require the probability of each  $\tau_{y,t}$  assignment, representing the prior probability of each possible component transition. Experience with *Livingstone* suggests that an order-of-magnitude probability scale is sufficient for two reasons. First, the internal behavior of a machine is usually far less stochastic than its interaction with its environment. There is an expected or nominal behavior that a component will exhibit for a given state and input. Failures are one or more orders of magnitude less likely. Second, precise estimates for these priors are often either inaccessible or unknown. In the case of spacecraft, the components may be unique or they may be destined for a new operating environment. However, the relative plausibility of each failure mode during operation can be elicited quite easily. In this work, we formalize and capitalize on these characteristics of the priors by making use of infinitesimals (Goldszmidt & Pearl 1992) to model the relative likelihoods of failures.

An infinitesimal probability is represented by an infinitesimally small constant raised to an exponent referred to as the *rank*. The rank can be considered the degree of unbeliability. Intuitively, one would not consider a rank 2 infinitesimal believable unless all rank 0 and rank 1 possibilities had been eliminated. Composition of infinitesimals has many desirable properties. If  $A$  and  $B$  are independent events, then

$$\begin{aligned} \text{Rank}(AB) &= \text{Rank}(A) + \text{Rank}(B) \\ \text{Rank}(A \vee B) &= \min(\text{Rank}(A), \text{Rank}(B)) \end{aligned}$$

Thus an outcome that can occur through multiple independent events has rank  $i$  if one event has rank  $i$  and the remaining events, even if arbitrarily many, have ranks of  $i$  or more. This property is key. It allows us to consider only the most likely trajectories leading to a state: if a sequence of events of rank  $i$  ends in state  $s_j$ , then an arbitrary number of higher rank (i.e. less likely) trajectories leading to  $s_j$  will not change the of  $s_j$ . Similarly, if state  $s_j$  is reached by a trajectory of rank  $i$ , and no trajectory of rank  $i$  or less reaches  $s_k$ , then  $s_j$  is more likely than  $s_k$ . We need not consider the possibility that a vast number of unlikely trajectories lead to  $s_k$  and together increase its likelihood above that of  $s_j$ . Thus  $\Gamma(\tau_{y,t} = \tau^*)$  returns the rank of the likelihood of that assignment. We frame our algorithms in terms of most likely trajectories, knowing there is a direct correspondence to most likely

states given the infinitesimal interpretation of the priors.

### 3.5 Correspondence to the POMDP Formulation

The correspondence between a domain specified as a POMDP and a problem  $P$  specified as a transition system is straightforward. The state set  $S$  of  $P$  is the subset of the cross product of the variables of  $\Sigma$  that is consistent with  $\mathcal{M}_\Sigma$ . Similarly, a set of a system-wide actions  $\mathcal{A}$  must be formed from the the factored commands  $\mathcal{C}$ . If the transition system is limited to receiving one command per time step, then the action set  $\mathcal{A}$  is formed by considering each possible value for each command, and augmenting it with the idle value for all other commands. If the model-based controller may issue commands in parallel, then  $\mathcal{A}$  consists of the consistent cross-product of the command values. The observation set  $\mathcal{O}$  consists of the subset of the cross product of the variables  $\mathcal{O}$  that is consistent with  $\mathcal{M}_\Sigma$ .  $\mathcal{M}_\Sigma$  and  $\mathcal{M}_T$  provide a very compact encoding for the observation and transition functions  $O, T$ . Let  $s$  be a state of the POMDP we are constructing to correspond to the transition system  $P$  and  $o$  be an observation. Note that  $s$  is assignment to all variables in  $\Sigma$  and  $o$  is an assignment to the variables of  $\mathcal{O}$ . For a state  $s$  and an observation  $o$ , the POMDP observation function  $O$  is as follows, where assignment of a set of values to a set of variables simply assigns each value to the corresponding variable:

$$\begin{aligned} O(s, o) &= 1 && \text{if } (\Sigma = s) \wedge \mathcal{M}_\Sigma \models (\mathcal{O} = o) \\ O(s, o) &= 0 && \text{if } (\Sigma = s) \wedge \mathcal{M}_\Sigma \wedge (\mathcal{O} = o) \models \perp \end{aligned}$$

That is, if the assignment of the state variables in the transition system model entails the observation assignment, the probability of the observation given the state in the corresponding POMDP is one. If the observation assignment and state assignment are inconsistent given the transition system state model, the probability is zero. The question arises as to the value of  $O(s, o)$  if  $(\mathcal{O} = o)$  is neither inconsistent with nor entailed by  $(\Sigma = s) \wedge \mathcal{M}_\Sigma$ . This issue generally arises when using model-based diagnosis algorithms and is not an issue with viewing the problem as a POMDP per se. Often the choice is to make  $O(s, o)$  uniform over the consistent values of  $\mathcal{O}$ . In other cases, algorithms are constructed to implicitly model  $O(s, o) = 1$  in the absence of any other information.  $O(s, o)$  is then no longer a probability

distribution, as it sums to more than one when marginalized on  $s$ .

The transition function  $T$  is similarly specified.  $T(s, a, s')$  is the probability of the assignment to  $\mathcal{T}_i$  that transitions the system from  $s$  to  $s'$ , or 0 if no such consistent assignment exists. Given the independence assumptions, the probability of an assignment to  $\mathcal{T}_i$  is simply the product of the probabilities of the individual assignments to each variable. Recall that the belief state update algorithm for a POMDP is derived from the Markov property and Bayes' rule, and is specified by the following formulas:

$$\begin{aligned} p(s', b) &= \sum_{s \in \mathcal{S}} T(s, a, s') b(s) \\ b'(s') &= \frac{O(s', o) p(s', b)}{\Pr(o \mid a, b)} \end{aligned}$$

In the next chapter, we consider a number of belief state generation algorithms for the transition system formulation. These algorithms at their core use the same belief state update algorithm, but take advantage of the structure of  $\mathcal{M}_\Sigma$  and  $\mathcal{M}_\mathcal{T}$ . First, we need not consider any state that is inconsistent with the observations. That is, we may disregard any  $s'$  such that  $s' \wedge \mathcal{M}_\Sigma \wedge o \models \perp$ , as  $O(s', o) = 0$  and thus  $b'(s') = 0$ . If a partial assignment to  $\Sigma$ ,  $c$ , is such that  $c \wedge \mathcal{M}_\Sigma \wedge o \models \perp$ , then all states  $s' \in c$  can immediately be eliminated from consideration. The partial assignment  $c \cup o$  is referred to as a *conflict* and *conflict-directed* searches use the conflicts they have discovered to greatly reduce the number of states they must examine. Second,  $T(s, a, s')$  is the product of the independent probabilities of assignments to members of  $\mathcal{T}_i$ . Best-first search techniques take advantage of the compositionality of a trajectory's probability to construct assignments to  $\mathcal{T}_i$  so as to consider transitions  $T(s, a, s')$  in order of probability. Together, these two observations will allow us to construct algorithms that incrementally generate the non-zero members of the belief state in order of probability.

## Chapter 4

# Trajectory Tracking Algorithms

### 4.1 The *CBFS-track* Trajectory Tracking Algorithm

The transition-system formulation suggests an intuitive procedure to begin enumerating the belief state at any point. The transition system is initialized with  $\mathcal{M}_\Sigma$  and a copy of all variables, representing the initial state. At time step  $t$ , we introduce the structure needed to represent the feasible next states of the system. We first create a copy of all variables and extend  $\mathcal{M}_\Sigma$ , conceptually introducing a new copy of the state constraints where the variables have new time indices, to model the constraints between the variables within the new time step. We then extend  $\mathcal{M}_\mathcal{T}$ , representing the constraints between the variables in the current and next time steps. Finally, we assign  $\mathcal{C}_t$  and  $\mathcal{O}_{t+1}$  according to how the system was commanded and the observations that resulted.

**Example 5** Figure 4.1 illustrates a trajectory-tracking problem of length three for the model of Example 2.2. Each box represents an variable. The command is *cmdin* and the observations are *flow<sub>v1</sub>* and *flow<sub>v2</sub>*. These variables are assigned by the problem, as is the start state. The highlighted  $\tau_{y,t}$  assignments must be chosen. The remaining variables will be constrained based upon these assignments. The arcs represent constraints from  $\mathcal{M}_\mathcal{T}$ . Constraints from  $\mathcal{M}_\Sigma$  are not shown. For all  $\tau_{y,t}$  we will assume  $\text{Rank}(\tau_{y,t} = \textit{nominal}) = 0$  and  $\text{Rank}(\tau_{y,t} \neq \textit{nominal}) = 1$ .

Trajectories may be enumerated in order by enumerating assignments to all  $\tau_{y,t}$  in order of the sum of the ranks, then testing for consistency with  $\mathcal{M}_\mathcal{T}$  and  $\mathcal{M}_\Sigma$ . Conflict-directed, best-first search, or *CBFS* (Dressler & Struss 1992; de Kleer & Williams 1989; Williams



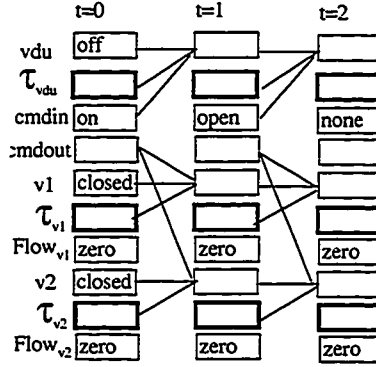


Figure 4.1: Evolution of the VDU/valve system

```

proc CBFS-track()
  problem  $X = \Sigma_0 \cup \mathcal{D}_0$ 
  Assign  $\Sigma_0$  to initial state;
  loop
     $\bar{X} = X \cup \mathcal{T}_t \cup \mathcal{C}_t \cup \Sigma_{t+1} \cup \mathcal{D}_{t+1}$  to represent new time step  $t + 1$ ;
    Assign  $\mathcal{C}_t$  and  $\mathcal{O}_{t+1}$  according commands and observations received;
     $Result = n$  most likely consistent assignments to  $\mathcal{T}$  returned by  $CBFS(X, \mathcal{M}_{\mathcal{T}} \wedge \mathcal{M}_{\Sigma}, f)$ 
    report  $Result$ ;
  }
} CBFS-track

```

Figure 4.2: CBFS-based trajectory tracking algorithm

& Nayak 1996) greatly focuses this process by using conflicts. In this context, a conflict is a partial assignment to  $\mathcal{T}$  and  $\mathcal{O}$  that is inconsistent. When a candidate solution is found to be inconsistent, the conflict is recorded in a database, ConflictDB. No further candidates that contain a known conflict are generated.

We begin with a representation of the initial state of the system in  $X$ . At each time step, we extend  $X$  with the variables necessary to represent the transition to the next state. We then assign  $\mathcal{C}_t$  and  $\mathcal{O}_{t+1}$  according to how the system was commanded and the observations that resulted. *CBFS* is then used to enumerate consistent assignments to  $\mathcal{T}$  in best-first order. The enumeration could be continued until  $n$  assignments are found, until the rank of the next assignment found decreases, or until some other stopping criterion computable from the solutions found thus far is met. The process is then repeated for the next time step.

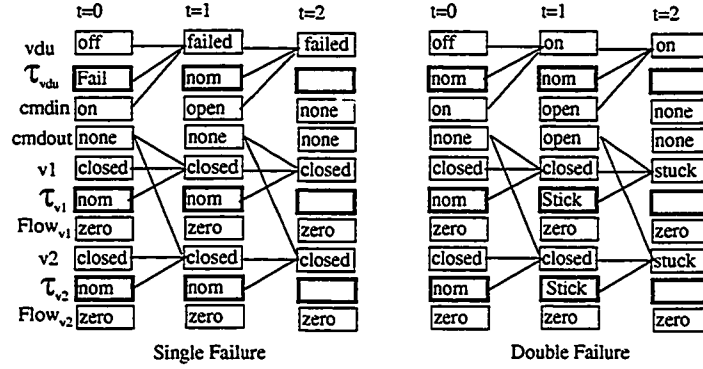


Figure 4.3: Two evolutions of the system

At each time step, this procedure recomputes from scratch the most likely assignments to the transition variables given all observations. A trajectory  $t$  that was most likely given only previous observations might only be consistent with a new observation if it is extended to  $t'$  by a very unlikely assignment to the most recently added transition variables. Since *CBFS-track* reconsiders all transition assignments at each time step in best-first order,  $t'$  will only be considered if there are no consistent trajectories whose probability lies between  $t$  and  $t'$ . Thus at all time points *CBFS-track* recomputes the most likely members of the belief state given all available commands and observations.

**Example 6** Figure 4.3 illustrates the two lowest-cost solutions to the above problem that would be found by *CBFS*. They represent a single failure of rank 1 at time 1 and a double failure of rank 2 at time 2, respectively.

While this algorithm does compute the most likely states at each time step, it has several significant drawbacks. First, the memoryless quality that allows *CBFS-track* to avoid over committing to seemingly likely trajectories also forces the algorithm to rediscover past failures at each time step the system is tracked. Since failures are the exception rather than the rule, we would like a tracking procedure that minimizes computation when no failures have occurred, and when failures do occur, scales the computation required with the inverse of their likelihood. More importantly, the size of the *CBFS* problem to be solved is very large for any given trajectory length and grows unboundedly as the trajectory is extended over time. *CBFS-track* does nothing to eliminate variables within a time step based upon

the structure of the problem, nor does it attempt to truncate the trajectory representation to maintain a bounded problem size.

## 4.2 The *Livingstone* Algorithm

The *Livingstone* system (Williams & Nayak 1996) uses *CBFS* to perform both state identification and control. In this section we will focus on the use of *CBFS* for state identification. While *Livingstone* was not developed from exactly the transition system formalism described in this paper, it seeks to solve the same underlying problem and can be described in this framework.

In order to avoid the problem of an ever increasing number of variables to assign that *CBFS-track* encounters, *Livingstone* does not track the most likely states or trajectories given the commands and observations thus far. It instead approximates the problem of tracking the most likely trajectories given all available information as a recurring trajectory tracking problem of length one. In this problem, the current state is assumed to be known or to be contained in a small set, and the task is to identify the most likely next states given the assignments to  $\mathcal{C}_t$  and  $\mathcal{O}_{t+1}$ . The true current state is then assumed to be in the set of currently most likely states, and the problem recurs. The *Livingstone* algorithm solving this problem is illustrated in Figure 4.4. The parameter  $n$  specifies how many states *Livingstone* will track. *Livingstone* in fact represents a class of algorithms that solve the following problem: Given that the system was in one of  $n$  states  $m$  time steps ago, determine the most likely state given the intervening commands and observations, and then use this as an approximation for the most likely portion of the belief state given the entire trajectory. The current *Livingstone* implementation sets  $n = 1$  and  $m = 1$  in order to solve problems such as Figure 1.3 with sub-second response time using the relatively weak CPU's found on spacecraft (Bernard *et al.* 1998).

*Livingstone* does not share the commitment-less property of *CBFS-track* in that it does not reconsider all transition assignments at each time step. It is still the case that a trajectory  $t$  that was most likely given only previous observations might only be consistent with a new observation if it is extended to  $t'$  by a very unlikely assignment to a transition variable. *Livingstone* considers only the newest transition assignment, in essence committing to all

```

proc Livingstone(n)
  problem  $X = \Sigma_0 \cup \mathcal{D}_0 \cup \mathcal{T}_0 \cup \mathcal{C}_0 \cup \Sigma_1 \cup \mathcal{D}_1$ 
  for ( $i=0; i < n; i=i+1$ )
     $Worlds_i$  = initial state;
  }
  loop
    Assign  $\mathcal{C}_0$  and  $\mathcal{O}_1$  according to new commands and observations received;
    for ( $i=0; i < n; i=i+1$ )
       $\Sigma_0 = Worlds_i$ 
       $\mathcal{T}_0$  = most likely consistent assignment to  $\mathcal{T}_0$  returned by  $CBFS(X, \mathcal{M}_{\mathcal{T}} \wedge \mathcal{M}_{\Sigma}, f)$ 
       $Worlds_i = \Sigma_1$  as entailed by  $\Sigma_0 \wedge \mathcal{C}_0 \wedge \mathcal{T}_0 \wedge \mathcal{M}_{\mathcal{T}}$ 
    }
    Report  $Worlds$ 
  }
} Livingstone

```

Figure 4.4: Livingstone trajectory tracking algorithm

previous assignments. There is no choice but to extend  $t$  to  $t'$ , even if an assignment of greater likelihood than  $t'$  could be found by reconsidering the assignments represented by  $t$ . Thus *Livingstone* does not track the most likely states of a system. Rather, *Livingstone* determines the  $n$  most likely successors, given the next set of commands and observations, of the states that were the most likely successors given the previous set of commands and observations. Thus each partial belief state is made up of only descendants of the previous approximation of the  $n$  most likely states, which were determined using only previous observations. This approach is tractable but fairly vulnerable to ambiguity. Consider the following examples.

**Example 7** Recall the VDU system from Figure 2.2. In the first time step we turn the VDU on. From that point onward, the VDU may be on or failed. The only way to distinguish between the two is to attempt to command the valves. Before performing that action, it is much more likely that the VDU is on. *Livingstone* therefore commits to the state wherein the VDU is on. The next state of the system must now evolve from the state wherein the VDU is on. Suppose we now command the valves to open, and receive the observations that there is no flow at either valve. All evolutions from the state wherein the VDU is on and the valves are being commanded open to a state where there is no flow involves all of the valves moving to the stuck state. This holds whether we have two valves or

one hundred. Clearly if we had considered the entire trajectory of commanding the VDU then commanding the valves, then conditioning on the multiple flow observations, the most likely trajectory would involve the single failure of the VDU command.

**Example 8** Consider a computer that can fail in two ways: its software can hang, in which case it needs to be reset, or its hardware can hang, in which case it needs to be power cycled. Software hangs are significantly more likely than hardware hangs. In either case, the computer fails to respond to keyboard input. Suppose we receive the observation that the computer is not responding to input. It is ambiguous as to which failure the computer is experiencing, so *Livingstone* commits to the software hang. If a reset command fails to revive the computer, *Livingstone* will search for the most likely transition given that the computer was experiencing a software hang, and a reset command failed to revive it. Given the prior probabilities, the most likely failure is a software failure. Since *Livingstone* always considers the most likely extension to the current trajectory, rather than the globally most likely trajectory, there is never an opportunity to consider the hardware failure. *Livingstone* will thus continually consider a software failure, reset, and consider a software failure.

### 4.3 The Conflict Coverage Algorithm

Note that unlike *CBFS-track* or *Livingstone*, this algorithm should neither rediscover previous failures nor irrevocably commit to a trajectory or set of trajectories that are most likely given the only the current observations. If properly constructed, our procedure will have the following properties:

- It tracks all consistent trajectories at the most likely probability level.
- As long as trajectories at the current probability level remain, very little computation is required.
- As soon as it's no longer consistent to believe the system is in a state at the current probability level, the procedure finds and begins tracking all trajectories at the next probability level.

- Conflicts discovered at each probability level are accrued, ensuring that future conflict-directed searches are highly focused and do not reconsider trajectories that have previously been ruled out.

The strengths of efficiently tracking a partial belief state are merged with the flexibility of incrementally enumerating belief states in the *CoverTrack* procedure of Figure 4.5. *TSet* is a superset of all consistent trajectories of rank  $\gamma$ , as returned by a previous call to *CoverTrack*. As described above, *extend* adds to the transition system the variables needed to represent the outcomes of the current command. All trajectories are augmented by the new transition variables, which are assigned nominal transition, and checked for consistency. Any inconsistent trajectory requires additional failures above rank  $\gamma$ , and is discarded as relatively implausible. The survivors are a superset of all consistent trajectories of rank  $\gamma$ . If this set is not empty, it is returned. Otherwise, the most likely trajectory has a rank greater than  $\gamma$ . The *GenerateCover* algorithm generates all assignments to  $T$  of a given rank that cover all known conflicts. A conflict is covered if at least one of the variables in the conflict is assigned to an assignment that does not appear in the conflict. Intuitively, we leave the  $\tau_{y,t}$  at their zero rank values, introducing reassignment only to avoid conflicts, with a total cost of  $\gamma$ . This is the NP-hard *hitting set* problem. The contents of *ConflictDB* and  $\gamma$  will determine whether this problem is tractable. Because of the loss of observations at past time points, *GenerateCover* returns superset of all consistent rank  $\gamma$  trajectories. If at least one trajectory is consistent with the current observations, it is returned. If not,  $\gamma$  is increased.

#### 4.4 Additional Related Work

A more inclusive synthesis of the literature on belief revision and belief update was performed by Friedman and Halpern (Friedman & Halpern 1999). It describes a general, plausibility-based temporal logic framework for describing belief revision methods, into which our work can be placed. The trajectory tracking method described here differs from that described by Friedman and Halpern and the other approximations of which the authors are aware in that it uses history to compensate for not having a sufficient statistic.

```

proc CoverTrack(cmd, obs, TSet, ConflictDB,  $\gamma$ ) {
  /*Extend the system adding  $\Sigma_t$  to  $\Sigma$ ,  $T_t$  to  $T$ */
  extend( $\Sigma$ ,  $T$ , cmd);
  /*Extend trajectories at current  $\gamma$  */
  Assign  $T_t$  to nominal, 0 rank assignment.
  for trajectory in TSet
    trajectory = trajectory  $\cup T_t$ ;
  /*Check trajectories for consistency, up  $\gamma$  if needed*/
  Assign  $\mathcal{O}$  according to obs received;
  Survivors =  $\emptyset$ ;
  loop{
    for trajectory in TSet {
      conflict=checkConsistency(trajectory);
      if (conflict) then
        push(conflict, ConflictDB);
      else
        push(trajectory,survivors); }
    if(survivors) then return survivors;
    /*Ran out of trajectories. Find more at next rank*/
     $\gamma = \gamma + 1$ ;
    TSet=GenerateCover( $T$ ,ConflictDB, $\gamma$ );
  }
}

```

Figure 4.5: Conflict Coverage Tracking Procedure

## Chapter 5

### Decreasing the problem size

While applying CBFS or *CBFS-cover* to the full transition system exactly enumerates the most likely trajectories, and thus states, in order, problem size is a significant issue. Let  $p$  denote the number of propositions needed to represent each possible value of each variable in  $TU\Sigma UCUDUO$ . These propositions are constrained by a copy of  $\mathcal{M}_{\mathcal{T}}$  and  $\mathcal{M}_{\Sigma}$  at each time step. Testing consistency of an  $m$ -step candidate trajectory is a consistency problem of  $m \times p$  propositions and  $m \times |\mathcal{M}_{\mathcal{T}} \cup \mathcal{M}_{\Sigma}|$  clauses. For the Deep Space 1 model, this is  $m \times 4041$  propositions and  $m \times 13,503$  clauses.

Let  $v$  be the number of ways to choose a variable from  $\mathcal{T}$  and assign a failure value ( $rank > 0$ ) value to it. There are  $m \times n$  variables in  $\mathcal{T}$ . Let  $f$  denote the average number of failure assignments per variable. Thus,  $v = m \times n \times f$ . To find the most likely consistent candidate assuming a single failure, the number of consistency checks that would have to be performed on this large theory would be  $\mathcal{O}(v)$  in the worst case: any  $\tau_{i,t}$  could be assigned to any failure value in its domain. Finding an arbitrary combination of failures would require a number of consistency checks exponential in  $v$ .

In this chapter, we reduce the structure needed to represent the evolution of the system at a time point from a complete copy of the system model to a small number of variables and clauses. Intuitively, when a command is issued to the system, only a small number of components participate in transmitting that command through the system or transitioning in response to the command. Consider Figure 5.1. The squares represent state variables, the lines sets of constraints from  $\mathcal{M}_{\mathcal{T}}$ . As of time 7, the valves, pump and VDU have not



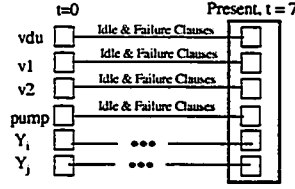


Figure 5.1: Evolution before commanding the valves

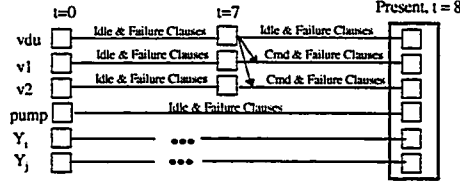


Figure 5.2: Evolution upon commanding the valves

been commanded nor have they interacted with other components by passing a command. If we had not detected a failure of any of these components, we can represent the possibility that they remained idle or failed in a localized and unobservable way with a single set of variables and constraints as illustrated. At time 7 we command the valves on. We require variables  $v1_8$  and  $v2_8$  to represent the new states of the valves.  $\mathcal{M}_{\mathcal{T}}$  suggests  $vdu_7$ ,  $v1_7$  and  $v2_7$  will interact with  $v1_8$  and  $v2_8$ . These variables, along with necessary transition variables  $\tau_{vdu,7}$ ,  $\tau_{v1,7}$  and,  $\tau_{v2,7}$ , are introduced to the system with the appropriate clauses from  $\mathcal{M}_{\mathcal{T}}$ . For each other variable  $y$ , the variable representing  $y_7$  is adequate to represent  $y_8$ . Figure 5.2 illustrates this process. In order to derive a well-founded algorithm from these intuitions, we first place a natural restriction on  $\mathcal{M}_{\mathcal{T}}$  that does not impact correctness. Second we introduce an approximation involving  $\mathcal{M}_{\Sigma}$  that, importantly, does not rule out consistent trajectories. Instead, some trajectories that are not consistent with past observations may be admitted, with the possibility that future observations will eliminate them. These problem modifications avoid replication of many variables in  $\Sigma$  and  $\mathcal{D}$ , as well as corresponding constraints from  $\mathcal{M}_{\mathcal{T}}$  and  $\mathcal{M}_{\Sigma}$ .

### 5.1 Restricting $\mathcal{M}_{\mathcal{T}}$

We restrict  $\mathcal{M}_{\mathcal{T}}$  in the same manner *Livingstone and Burton* (Williams & Nayak 1997): a component moves to a failure state with equal probability from any state, and, except for

failures, a component that does not receive a command idles in its current state.  $\mathcal{M}_{\mathcal{T}}$  is limited to the forms:

$$\begin{aligned} (\tau_{y,t} = \tau_{fail}) &\implies y_{t+1} = y_{fail} \\ (\mathcal{C}_{y,t} = C^*) \wedge \phi_t \wedge (\tau_{y,t} = nom) &\implies y_{t+1} = y^* \\ (\mathcal{C}_{y,t} = idle) \wedge (\tau_{y,t} = nom) &\implies y_{t+1} = y_t \end{aligned}$$

where  $\phi_t$  is a propositional formula over  $\Sigma_t \cup \mathcal{D}_t$ ,  $C^* \in \delta(\mathcal{C}_{y,t})$ ,  $nom \in \delta(\tau_{y,t})$  and  $\tau_{fail} \in \delta(\tau_{y,t})$ . Formulas of the first form model failures while formulas of the second form model nominal, commanded transitions. Formula of the third form are frame axioms that encode our assumption that devices that do not receive a command remain in their current state. We replace  $\phi_t$  with implicant  $\pi_t$ , an equivalent formula involving only  $\Sigma_t$ . Intuitively  $\phi_t$  is a formula involving  $\mathcal{D}$  that, given  $\mathcal{M}_{\Sigma}$  and an assignment to  $\Sigma$ , allows us to infer if  $\mathcal{C}_{y,t}$  propagates through a set of components to component  $y$ . To form  $\pi_t$ , we replace each assignment to  $\mathcal{D}_t$  with a set of assignments from  $\Sigma_t$  that imply the  $\mathcal{D}_t$  assignment under  $\mathcal{M}_{\Sigma}$ . We expect that for the type of clauses  $\mathcal{M}_{\mathcal{T}}$  contains, growth in  $\pi_t$  will be proportional to the length of the component chain that transmits  $\mathcal{C}_{y,t}$ , which ranged from 1 to 5 in (Bernard *et al.* 1998). Our experience supports this hypothesis. This growth is offset as non-idle, non-failure clauses take the following form which is independent of  $\mathcal{D}$ :

$$(\mathcal{C}_{y,t} = C^*) \wedge \pi_t \wedge (\tau_{y,t} = nom) \implies y_{t+1} = y^*.$$

Given a  $\mathcal{C}_{y,t}$  which is not idle, in order to determine consistency with  $\mathcal{M}_{\mathcal{T}}$  we now need only introduce  $\mathcal{C}_{y,t}$ ,  $\tau_{y,t}$  and those select members of  $\Sigma_t$  that appear in  $\pi_t$ .

## 5.2 Eliminating intermediate observations

$\mathcal{M}_{\Sigma}$  remains, and requires introduction of all variables in  $\Sigma_t$  and  $\mathcal{D}_t$  in order to check consistency against  $\mathcal{O}_t$ . We proceed by eliminating all variables  $\mathcal{O}_t$  for values of  $t$  sufficiently far in the past. That is to say, transition choices are only constrained by consistency between the trajectories they imply and recent observations. As the system evolves, variables representing older observations and the copies of  $\mathcal{M}_{\Sigma}$  that constrain them are unneeded. For the portions of the trajectory where  $\mathcal{M}_{\Sigma}$  is not introduced, we need not introduce  $\mathcal{D}$  and need only introduce the limited portion of  $\Sigma_t$  required by  $\mathcal{M}_{\mathcal{T}}$ . This is of course an approximation. It is now possible to choose transition assignments that are inconsistent

with the discarded observations, resulting in an “imposter” trajectory. This approximation has several important features. First, it is a conservative approximation in that no consistent trajectories are eliminated. Second, all trajectories are checked against new observations, and impostors are eliminated as soon as they fail to describe the on-going evolution of the system. Finally if conflicts are recorded in *ConflictDB*, no partial assignment to  $\mathcal{T}$  that was discovered to be in conflict with the observations will be reconsidered, even after observations are discarded. Thus we can only admit an imposter in the case where a transition choice is in conflict with an observation, but the choice is not considered until after the conflicting observation has been discarded.

### 5.3 Selective Model Extension

Based upon these restrictions, the procedure *extend* introduces into time step  $t$  only the small fraction of the model involved with the evolution of the system due to the command  $C_{y,t} = C^*$ . Recall that the transition system model  $\mathcal{M}_{\mathcal{T}}$  is a set of formulas that for each component represent the possibility that the component has idled, has failed, or has transitioned based upon some command. The transition model for each commanded component has been compiled into a prime implicant form  $\pi_t$ . The intuition is that at a given time step components that are neither commanded nor influence the behavior of components that are commanded do not need to be explicitly represented. Thus the resulting problem size per time step is proportional to  $|\pi_t|$  for the commanded components. Conceptually, Theorem 1 allows us to determine which constraints can be eliminated without impacting the logical consistency of our transition system model. For the purpose of discussion we will assume that for each time step  $t$  there exists only one  $y$  for which  $C_{y,t} \neq \text{idle}$ . The proofs can be extended to parallel commanding.

**Theorem 1** Assume  $C_{y,t} = C^*$ ,  $C^* \neq \text{idle}$ , and for all  $x \neq y$ ,  $C_{x,t} = \text{idle}$ . Consider the formula of  $\mathcal{M}_{\mathcal{T}}$

$$(C_{y,t} = C^*) \wedge \pi_t \wedge (\tau_{y,t} = \text{nom}) \implies y_{y+1} = y^*.$$

For all state variables  $x_t$ ,  $x \neq y$ , if  $x_t \notin \pi_t$ , then an equivalent consistency problem is

formed by replacing  $x_t$ ,  $\tau_{x,t}$  and all formulas of  $\mathcal{M}_{\mathcal{T}}$  involving these variables with a constraint between  $x_{t-1}$  and  $x_{t+1}$ .

Intuitively, there are no witnesses to the value of  $x_t$  except for  $x_{t-1}$  and  $x_{t+1}$ , which can be constrained directly. If  $x_t$  is as described, then the only clauses involving  $x_t$  are of the form:

$$\begin{aligned}
(\mathcal{C}_{x,t-1} = \mathcal{C}^*) \wedge \phi_{t-1} \wedge (\tau_{x,t-1} = \text{nom}) &\implies x_t = x^* \\
(\mathcal{C}_{x,t-1} = \text{idle}) \wedge (\tau_{x,t-1} = \text{nom}) &\implies x_t = x_{t-1} \\
(\tau_{x,t-1} = \tau_{\text{fail}}) &\implies x_t = x_{\text{fail}} \\
(\mathcal{C}_{x,t} = \text{idle}) \wedge (\tau_{x,t} = \text{nom}) &\implies x_{t+1} = x_t \\
(\tau_{x,t} = \tau_{\text{fail}}) &\implies x_{t+1} = x_{\text{fail}}
\end{aligned}$$

The variable  $x_t$  can only impact the consistency of the system via the assignments to  $\tau_{x,t-1}$  and  $\tau_{x,t}$ . Given the independence assumptions, assigning failures to both is indistinguishable from and less likely than assigning  $\tau_{x,t-1} = \text{nom}$  and  $\tau_{x,t}$  to a failure, while assigning a failure to one is equivalent to assigning a failure to the other. Thus we need only consider  $\tau_{x,t-1} = \tau_{x,t} = \text{nom}$  and  $\tau_{x,t-1} = \text{nom}, \tau_{x,t} = \tau_{\text{fail}}$ . In the nominal case,  $x_t$  is equivalent to  $x_{t+1}$  and can be eliminated. In the failure case, the assignment to  $x_t$  has no impact on  $x_{t+1}$  and can be eliminated. The above formula are rendered equivalent to the following reduced set:

$$\begin{aligned}
(\mathcal{C}_{x,t-1} = \mathcal{C}^*) \wedge \pi_{t-1} \wedge (\tau_{x,t-1} = \text{nom}) &\implies x_{t+1} = x^* \\
(\mathcal{C}_{x,t-1} = \text{idle}) \wedge (\tau_{x,t-1} = \text{nom}) &\implies x_{t+1} = x_{t-1} \\
(\tau_{x,t-1} = \tau_{\text{fail}}) &\implies x_{t+1} = x_{\text{fail}}
\end{aligned}$$

In fact, at time  $t$  we will know whether or not  $\mathcal{C}_{x,t-1} = \text{idle}$ , and therefore we need only introduce one of the first two formulas. The *extend* procedure repeatedly applies the reformulation suggested by Theorem 1 to avoid introducing a variable or constraints for  $x_t$  when there have been no witnesses to  $x_t$  and it is possible to constrain  $x_{t+1}$  directly from

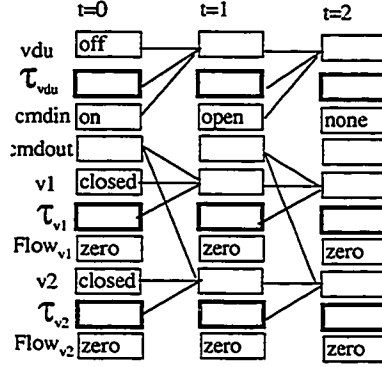


Figure 5.3: Expansion of the VDU Problem to Depth  $m = 3$ .

$x_{t-1}$ . When a command is introduced, the compiled  $\mathcal{M}_{\mathcal{T}}$  determines what clauses should be added to constrain the nominal transition of  $y_t$  under  $\mathcal{C}_{y,t}$ . State variables appearing in the introduced clauses are added, along with constraints representing their idle or failure transitions. By reducing the number of variables and clauses introduced at each time step, we reduce the consistency problem involved in checking a trajectory to a number of variables proportional to  $m \times |\pi_t|$ . The number of clauses is proportional to  $m \times (|\pi_t| + k)$  where  $k$  is the number of failure values per  $\tau_y$  domain.

## 5.4 Finite Horizons

While selective extension reduces the variables per time step, we still require an unbounded number of variables over time. We avoid this requirement by setting a finite horizon  $h$  steps in the past, beyond which all assignments are summarized by a single assignment. We summarize the  $l$  most likely assignments to all variables  $\tau_{y,t}$  where  $t < (m - h)$  into  $l$  different assignments to a single variable *History*. All other possible assignments to the initial  $\tau$  variables are discarded. The horizon point  $(m - h)$  is fixed relative to the present, and therefore only a bounded number of variables are required.

While a finite horizon is most useful when  $m$  has become large, we will illustrate the concept with a small example. Consider Figure 5.3. The VDU system has been tracked for 3 time steps and the model has been expanded accordingly. The variables  $\tau_{vdu,0}$  through  $\tau_{vdu,2}$  and  $\tau_{v1,0}$  through  $\tau_{v1,2}$  have been introduced to represent choices in the system's evolution.

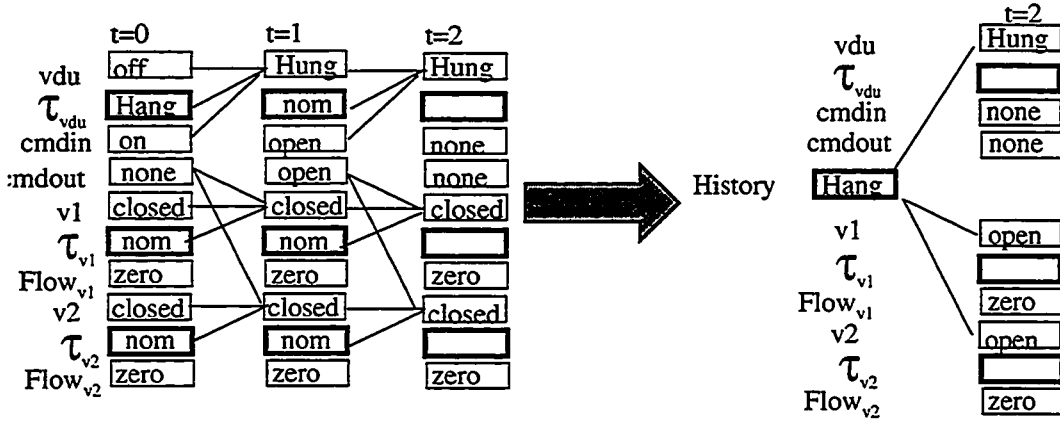


Figure 5.4: Summarization of the initial trajectory,  $\{\tau_{vdu,0}=\text{Hang}\}$

While tracking the system, we have incrementally generated the most likely consistent trajectories, represented by the most likely consistent assignments to all  $\mathcal{T}$ . Let us consider the case where the most likely trajectories given the observations thus far have been determined to be a VDU failure at  $t = 0$  or a double valve failure at  $t = 1$ . These are represented by the following assignments:

$$\{\tau_{vdu,0} = \text{Hang}, \tau_{vdu,1} = \text{nom}, \tau_{v1,t} = \text{nom}, \tau_{v1,1} = \text{nom}, \tau_{v2,t} = \text{nom}, \tau_{v2,1} = \text{nom}\} \quad (5.1)$$

$$\{\tau_{vdu,0} = \text{nom}, \tau_{vdu,1} = \text{nom}, \tau_{v1,0} = \text{nom}, \tau_{v1,2} = \text{Stick}, \tau_{v2,0} = \text{nom}, \tau_{v2,2} = \text{Stick}\} \quad (5.2)$$

Note that these are the most likely assignments to  $\tau_{y,0}$  and  $\tau_{y,1}$  given the observations received in the first three steps. Note that each assignment entails a set of values for  $\Sigma_2$ . For each likely full assignment, we can introduce a single variable assignment that summarizes the full assignment. Consider Figure 5.4. At the left of the figure, we have installed assignment 5.1, thus entailing values for the variables at  $t = 2$ . At the right, we have eliminated all variables at  $t = 0$  and  $t = 1$  and directly constrained the variables of  $t = 2$  from a new assignment,  $\text{History} = \text{Hang}$ . If we define  $\Gamma(\text{History}=\text{Hang})$  to be equal to the rank of assignment 5.1, then we have an equivalent representation of this trajectory with far fewer variables. Similarly, in Figure 5.5 we represent the trajectory of assignment 5.2 with the assignment  $\text{History} = 2\text{Stick}$ . Once we have summarized a sufficient number of trajectories (here two) with unique assignments to  $\text{History}$ , the variables at  $t = 0$  and  $t = 1$  are discarded.

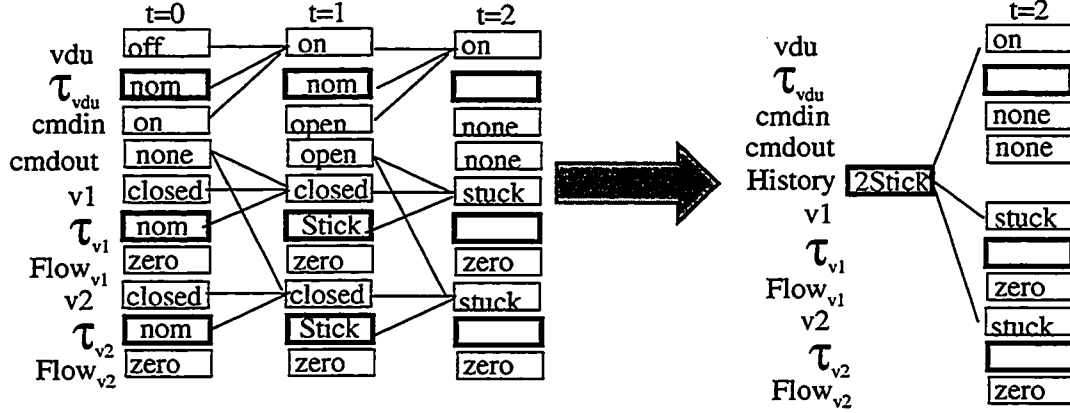


Figure 5.5: Summarization of the initial trajectory  $\{\tau_{v1,1}=\text{Stick}, \tau_{v2,1}=\text{Stick}\}$ .

We need not truncate the horizon after only two time steps. Imagine that we continue to expand the representation for  $m$  steps, conditioning on observations and finding the likely trajectories. Each  $m$ -step trajectory contains an initial segment that is most likely given a large number of observations received during time steps 0 through  $m$ . When  $m$  is sufficiently long to give us confidence that our most likely initial trajectories include the actual trajectory, or if we simply can only afford to store  $m$  steps, we apply the summarization. We replace each of  $l$  likely assignments to  $\tau_{y,0}$  through  $\tau_{y,(m-h)}$  with an assignment  $History = choice_i$  that has the same rank. We then entail the equivalent values in  $\Sigma_h$ . For each value  $y_h = y^*$  that was previously entailed by the non-truncated representation, we introduce a clause of the form  $History = choice_i \implies y_h = y^*$ .

This summarization may be applied repeatedly. If we expand the summarized representation of Figure 5.4 and Figure 5.5, we may then summarize the  $l$  most likely assignments to  $History$  and all  $\tau_{y,2}$  into a new variable  $History'$ . We may also summarize the oldest time step or several time steps leading to the oldest time step. By repeatedly applying the summarization as we extend the transition system, we can maintain a fixed size representation. The summary variable  $History$  restricts choices for the initial portion of the trajectory to the partial trajectories that appeared most likely after being extended for some time. Intuitively, we are trading the ability to represent an exponential number of initial trajectories with increasingly unlikely prior probabilities for a constant problem size and search space.

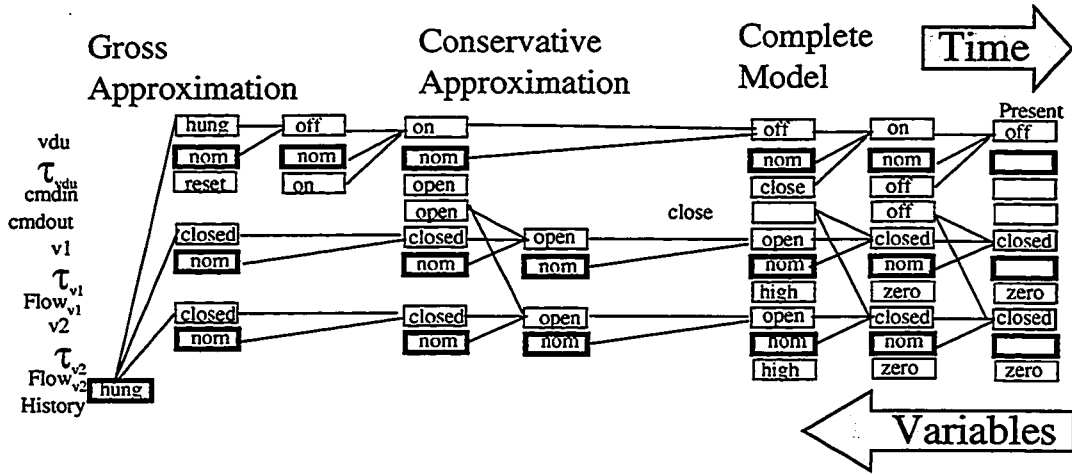


Figure 5.6: The Complete Representation

Unlike the previous approximation, this approximation is not conservative. If the true trajectory involves an assignment to an old  $\mathcal{T}$  variable that is not captured in a summarization, it is lost. This case can only arise when a failure too unlikely to be tracked has occurred, yet it was able to remain consistent with the observations until that part of the history was truncated.

Figure 5.6 shows a complete representation making use of both the conservative approximation and a finite horizon. At the right of the figure, new variables are introduced to represent the time steps. Here, where the trajectory assignments have not yet been conditioned on a large number of variables, we have a full model of the system. As variables age, they are moved into a conservative approximation. The assignments here have already been conditioned on observations within their time step before aging. They will now be conditioned upon how they effect the evolution of the system and whether they maintain consistency with incoming observations. Assignments that have both been conditioned on observations within their time step and later on how they impact newer observations, and have remained consistent, are summarized into the history variable. Thus the most space and search is reserved for the portion of the trajectory of which we are the least certain.



## Chapter 6

### Results for Diagnosis

We have implemented the transition system representation and the algorithms presented here in a software system called *L2* for Livingstone 2. In this chapter, we describe four *L2* applications, X-34, X-37, CB, ISPP, and Valves. All of these applications were developed by NASA engineers, except for Valves which was developed by the author. The X-34 and X-37 are two of NASA's experimental spacecraft programs. Experimental diagnosis systems for these spacecraft were developed at the NASA Ames Research Center. CB refers to a tutorial circuit breaker model developed by Charlie Goodrich at the NASA Kennedy Space Center. ISPP refers to an prototype in-situ propellant production plant also developed at NASA Kennedy Space Center. The Valves model is an instantiation of the tutorial problem presented in Figure 2.2. We first describe use of *L2* by NASA spacecraft engineers to illustrate how *L2* can be integrated into a control system and to convey the level of validation and testing the *L2* system has undergone. We then describe a set of experiments conducted by the author using the Valves, CB and ISPP models.

#### 6.1 Validation By NASA Spacecraft Engineers

*L2* has been used to develop experimental diagnosis systems for two NASA experimental spacecraft programs. The X-37, developed by Boeing, is shown in an artist's rendition in Figure 6.1. The X-37 was designed to be carried into orbit by the space shuttle, where it would orbit the earth for 21 days before autonomously landing. The X-34, developed by Orbital Sciences, is shown at NASA Dryden Flight Research Center in Figure 6.2. The

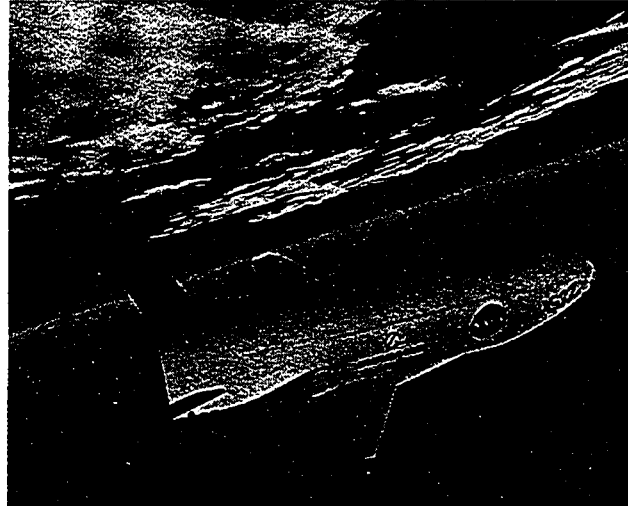
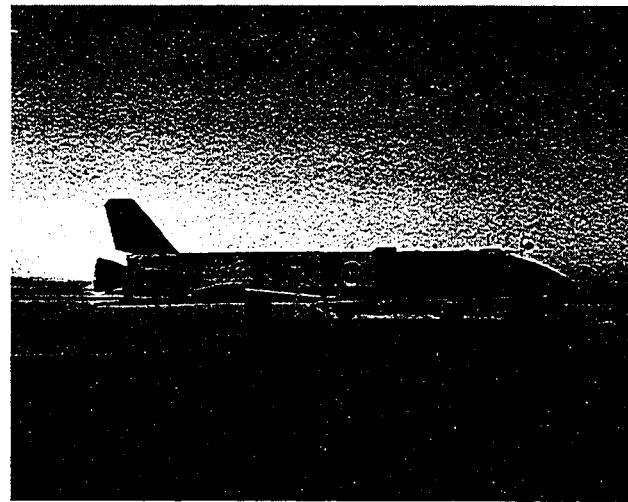


Figure 6.1: The X-37 Vehicle



NASA Dryden Flight Research Center Photo Collection  
<http://www.dryden.nasa.gov/gallery/photo/index.html>  
 NASA Photo: EC00-0226-7 Date: July 20, 2000 Photo by: Tom Tachada  
 X-34 on lakebed prior to tow tests

Figure 6.2: The X-34 Vehicle

X-34 was designed to be carried into the upper atmosphere where its rocket engine would ignite, propelling the craft to hypersonic speed. Figure 6.3 shows the typical deployment architecture for *L2*. The *L2* algorithms developed in the preceding chapters, labeled Livingstone in the Figure, are loaded with the spacecraft onto an embedded processor running a real-time operating system, typically a PowerPC running VxWorks. Sensors on the vehicle provide real-time input. Since *L2* reasons about discrete systems only, a set of software components called monitors abstract the continuous sensor input into a discrete space upon which the *L2* model is based. Diagnoses are generated based upon the commands and the discretized sensor input received by *L2* and either feed back into the vehicle's control system or reported to ground controllers.

For the X-37 application (Schwabacher, Samuels, & Brownston 2002), a model consisting 86 components was developed by NASA engineers to model the electro-mechanical portion of the craft, including power buses, power controllers, and electronic actuators for control surfaces. Twenty-eight failure scenarios were developed by the engineers to test the *L2* system and their models. In addition, the software was run on a real-time embedded computer for 21 days, the nominal mission duration. Unfortunately, the X-37 program was canceled for unrelated reasons before construction of the vehicle was completed, so no flight tests of this *L2* application was performed.

For the X-34 application (Bajwa & Sweet 2002), NASA engineers developed a model of the vehicle's propulsion system consisting of approximately fifty components, shown in Figure 6.4. This is the portion of the propulsion system that stores liquid oxygen (LOX) and rocket fuel, called RP1, and provides them to the rocket engine during thrusting. Figure 6.5 shows detail of the components used to model the LOX portion of the propulsion system. The large cannister-type objects are LOX tanks, the square icons represent valves, and the small circles are temperature and pressure sensors. Figure 6.6 illustrates the details of the RP1 portion of the system. Figure 6.7 lists a subset of the failure scenarios used to test *L2* and the X-34 model. The first column is the scenario number, while the second column is a text description of the failure. Each row in the fourth column is a diagnosis, listing the names of the failed components in the diagnosis. Column 5 lists the rank, based on the

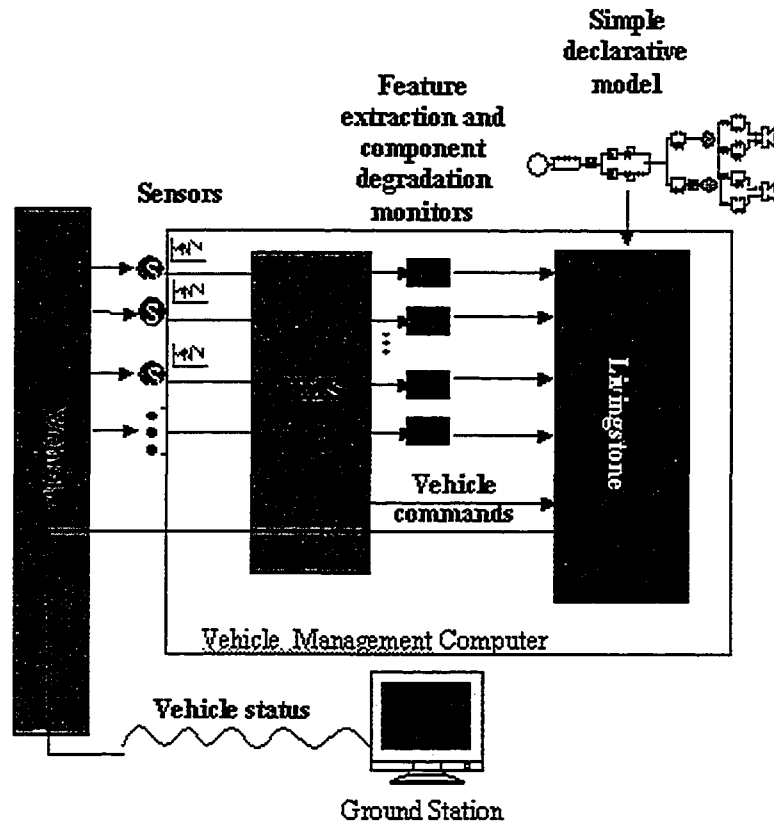


Figure 6.3: Typical L2 Deployment Architecture

use of the infinitesimal probability model described, for each diagnosis. Note that in many cases there are insufficient sensors to provide a unique diagnosis of the listed failure, so multiple diagnoses are generated and tracked. Unfortunately, the X-34 program was also canceled by NASA for unrelated reasons. *L2* was integrated with the necessary monitor and support code and tested on an embedded processor, using a simulation of the X-34 developed by NASA to provide continuous sensor data.

## 6.2 Experiments

In this section, we discuss a set of experiments performed by the author using the Valves, CB, ISPP models. *L2* is implemented in C++ in a modular form that allows alternative search and consistency procedures to be plugged into the transition system framework.

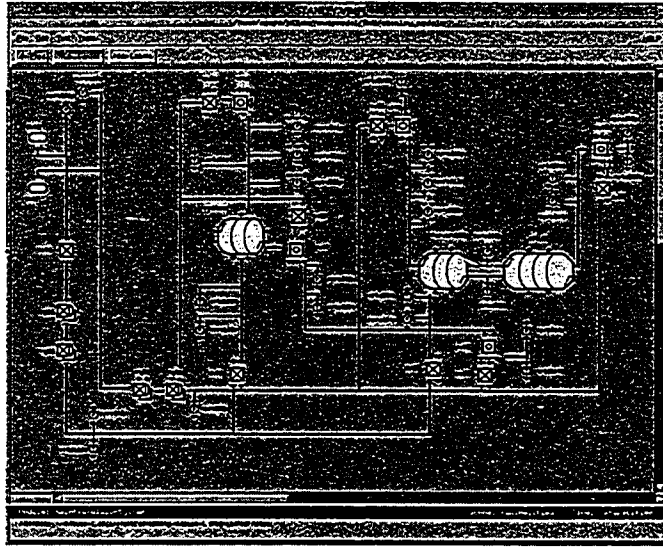


Figure 6.4: X-34 Model Schematic

These experiments were performed using *CoverTrack* for the search. Consistency was determined via a propositional consistency checker that used unit propagation and a truth maintenance system to cache inferences. Other consistency engines that allow use of integer or interval constraints and provide a complete inference mechanism can be developed. Use of unit propagation, which is an incomplete inference procedure in general, has no impact for the models presented in this chapter.

The embedded applications of *L2* described in the previous section use a very short horizon of length 2 to 5 in order to meet their real-time deadlines. In this section, an infinite horizon was selected, and the transition system structure was allowed to grow to lengths of up to approximately 700 time steps. Observations were applied to the model at only the time step representing the present. The tests were run under Windows NT on a 550MHz Pentium III.

### 6.2.1 Valves

The Valves model is an implementation of the valve and VDU model introduced in Chapter 1. It was developed simply to verify that *L2* correctly tracks the canonical scenarios known to confound *Livingstone*. We discuss it here simply to reinforce the *Livingstone2* diagnosis

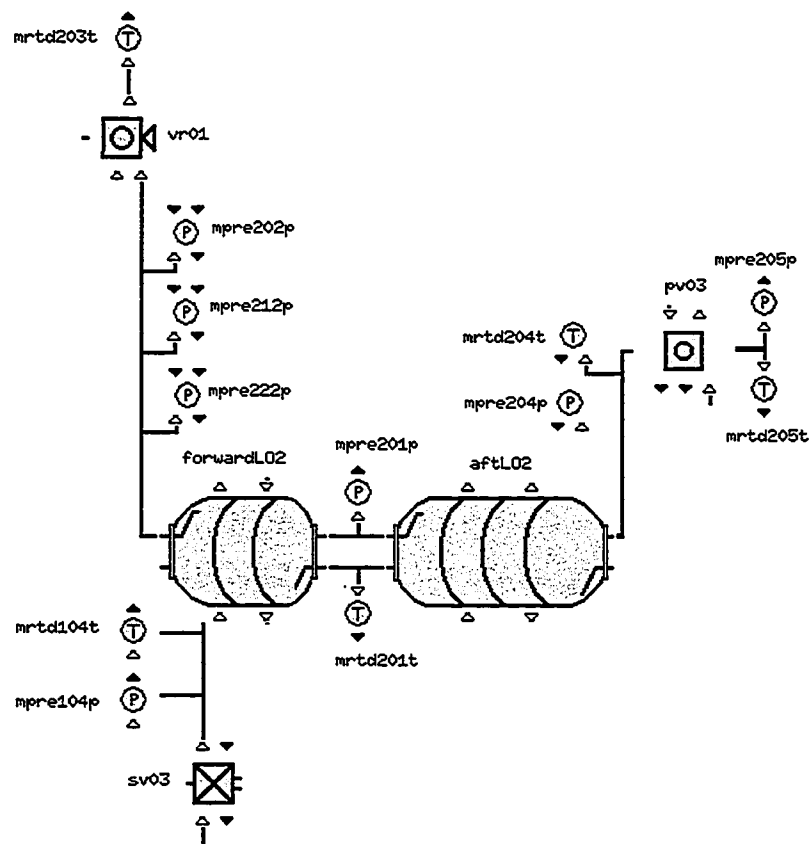


Figure 6.5: Schematic for the X34 LOX Subsystem

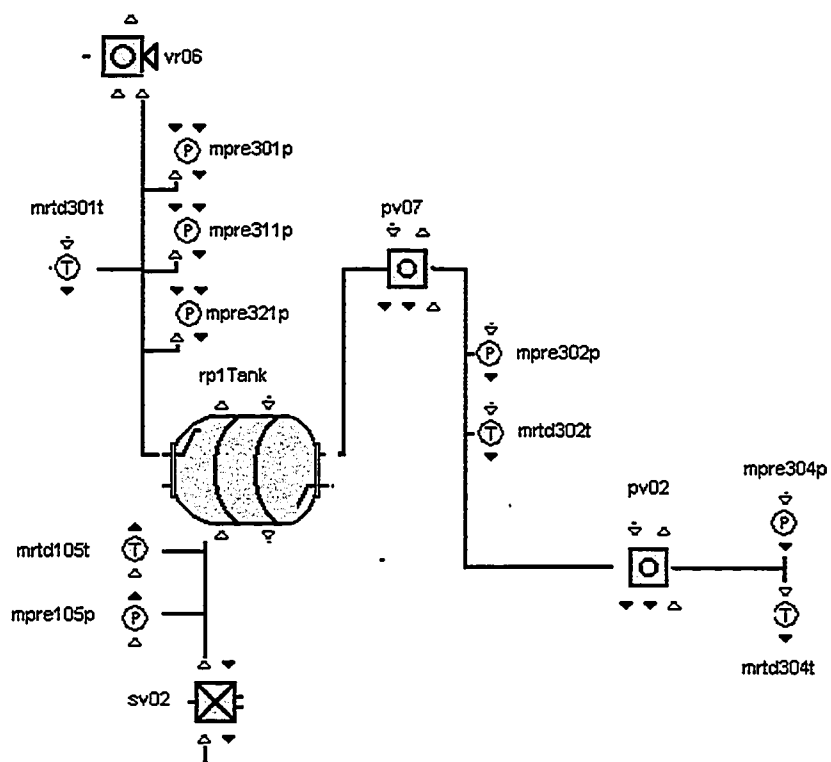


Figure 6.6: Schematic for the X34 Fuel Subsystem

PITEX Scenario	DESCRIPTION	TIME OF FAULT	DIAGNOSIS	Rank
1	Open microswitch, MMSW205X, fails on the LOX vent/relief solenoid valve, SV31.	3301.70 and 7260.8	Microswitch MMSW205X failed SV31 Failed Closed and LOX vent/relief valve VR01 Failed Open	3 4
2	Close microswitch, MMSW213X, fails on the LOX feed valve, PV03.	9410.00	Closed microswitch MMSW213X Failed	3
3	RP-1 feed valve, PV02, fails closed after the RP-1 bleed has been initiated.	9359.00	PV02 Stuck Closed SV32 Stuck Closed	2 2
4	RP-1 vent/relief pneumatic valve, VR06, fails open.	9379.00	VR06 Stuck Open MPRE103P faulty	2 2
5	Primary RP-1 tank pressurization valve, SV02, sticks closed.	9383.86	SV02 Stuck Closed SV02 Stuck Closed and SV02.openMicroswitch faulty SV02.openMicroswitch faulty and MPRE103P faulty SV02 unknown fault	2 5 5 5
6	Primary RP-1 tank pressurization valve, SV02, sticks open.	9384.71	SV02 Stuck Open SV02.openMicroswitch faulty SV02 unknown fault	3 3 5
7	Open microswitch, MMSW205X, fails on the LOX vent/relief solenoid valve, SV31. After that SV31 fails closed.	3301.70	SV31 Stuck Closed and SV31.openMicroswitch faulty VR01 Stuck Closed and SV31.openMicroswitch faulty	5 5
8	GHe pressurization system pressure regulators, RG11 and RG01, both regulate high.	9000.00	MPRE103P faulty RG11 regulates high and RG01 regulates high	2 4
9	Two of the LOX vent line pressure sensors, MPRE202P and MPRE212P, fail high.	0.00	MPRE202P faulty and MPRE212P faulty MPRE202P biased and MPRE212P biased MPRE202P faulty and MPRE212P biased MPRE202P biased and MPRE212P faulty	4 4 4 4
10	Two of the LOX vent line pressure sensors, MPRE202P and MPRE212P, fail low.	0.00	MPRE202P faulty and MPRE212P faulty MPRE202P biased and MPRE212P biased MPRE202P faulty and MPRE212P biased MPRE202P biased and MPRE212P faulty	4 4 4 4

Figure 6.7: X-34 Diagnosis Scenarios

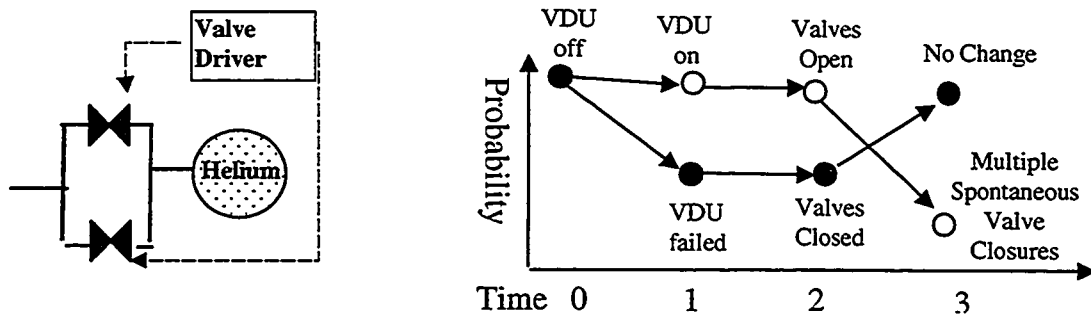


Figure 6.8: Evolution of a Valve Driver Unit and Valves



process. Figure 6.8 reintroduces the valve system. Recall that the helium tank pressurizes the system and the VDU commands the valves to open or close in parallel. Our canonical scenario was that at time 0 the VDU is off, the valves are closed and pressure is observed at the outlet of the helium tank. At time 0 the VDU is commanded on, but it fails. This failure is not immediately observable. At time 1, the VDU is commanded to open its valves. Since the VDU is failed, the valves do not change state. At time step 2, it is observed that there is no flow from any of the valves.

When the observations of no flow are received, *Livingstone* finds two conflicts in the current mode assignments: valve v1 cannot be open, and valve v2 cannot be open. Without the ability to revisit its assumption that the VDU was operating properly when the valves were commanded, the only explanation for the observed behavior is the incorrect diagnosis that all valves have failed simultaneously. The sequence of states *Livingstone* returns as its diagnoses are labeled as open circles in Figure 6.8.

Like *Livingstone*, *L2* initially tracks only the mode where the VDU is on at time step 1. However, its transition system data structure allows it to efficiently record that this state depends upon the assumption that the VDU did not fail at time step 0. When the valves are assumed to open at time step 2, their dependency upon the state of the VDU is also captured. Thus when *L2* receives the observations of no flow, it generates the following conflicts:  $\{\text{VDU}, v1\}$ ,  $\{\text{VDU}, v2\}$ . The lowest cost covering of this conflict set is to fail the VDU at time step 0 rather than to fail both v1 and v2. The initial trajectory being tracked is therefor discarded and trajectory shown in closed circles in Figure 6.8 replaces it.

Many more interesting scenarios were developed to check *L2*'s ability to incorporate new observations and correctly update the trajectories being tracked. We briefly outline two to illustrate the flexibility of this approach. Suppose the VDU is failed and only v1 is commanded open. We receive an unexpected observation of no flow only at v1, since v2 is expected to be closed. Assuming a valve failure is more likely than the VDU failure, the trajectory where v1 is stuck chosen over the trajectory where the VDU is failed. Now suppose v2 is later commanded, and no flow is reported at v2. Now lack of flow at v1 and v2 must be explained. The trajectory that contains just the failure of v1 is now no longer

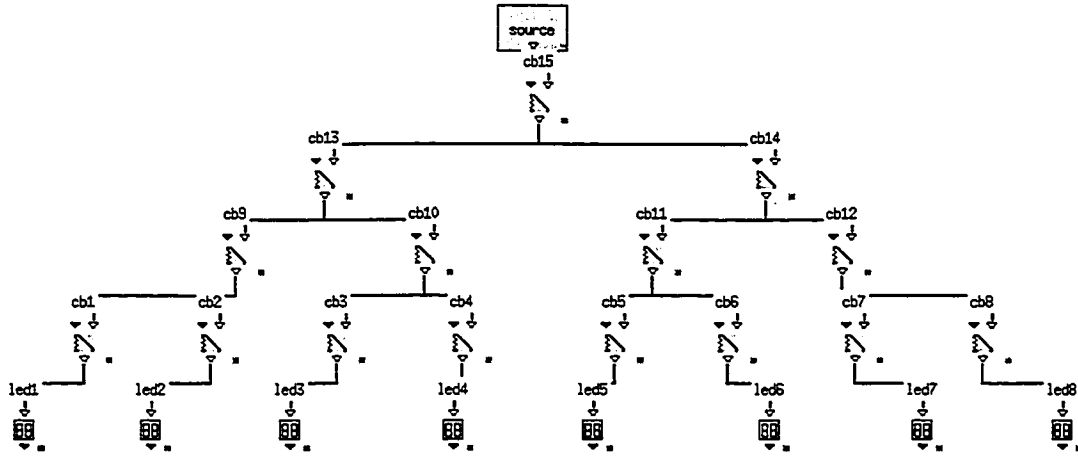


Figure 6.9: The Circuit Breaker (CB) Model

consistent. It will be extended or replaced. Let us assume two independent valve failures are less likely than the VDU failure. In this case, *L2* revisits its assumptions that *v1* has failed and the VDU has not. It finds the trajectory where the VDU had failed at step 0 and in fact *v1* never failed before considering that *v2* had failed in addition to the previously hypothesized failure of *v1*.

Consider if we extend the example so that the VDU has two failure modes. A likely software failure can be cleared by resetting the device, and a hardware failure is unlikely and permanent. When both valves are commanded and no flow observations result, the first trajectory to be tracked includes a software failure in the VDU. If the VDU is then reset in this state, it will move back to its nominal state. Suppose that the VDU is and the valves are commanded again. If there is still no flow, *L2* may find a trajectory where both valves were stuck all along, may find a trajectory that replaces the past resettable VDU failure with a permanent failure, or both, depending upon the ranks of the various failures.

### 6.2.2 CB and ISPP

Additional experiments were performed to investigate how the performance of *L2* would scale as the number of time steps being tracked in the trajectory increased. The Circuit Breaker, *CB*, model of 24 electrical components connected in series and parallel is illustrated in Figure 6.9. The *cb* components represent circuit breakers which can be turned

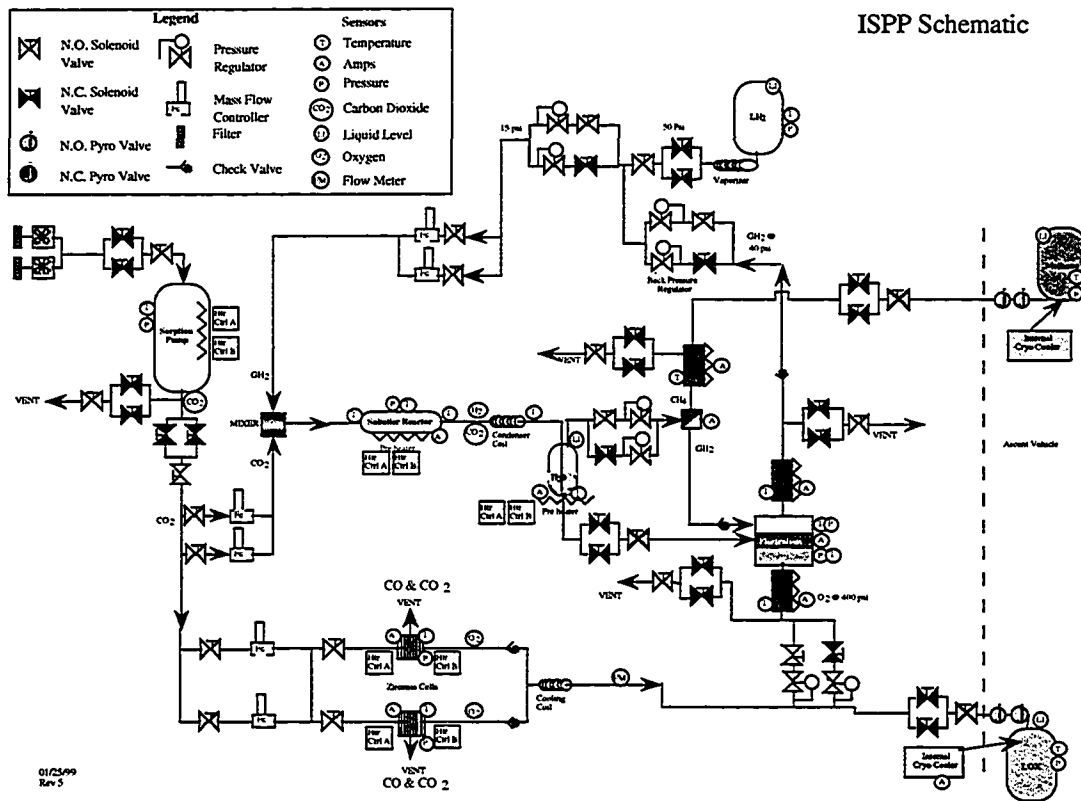


Figure 6.10: The In-Situ Propellant Production (ISPP) Model

on and off. They can also spontaneously fail into a tripped mode. Resetting a tripped circuit breaker returns it to its off mode. The led components represent light emitting diodes that light if there is a path of circuit breakers that are on between the diode and the power source. This model was developed as a tutorial by Livingstone user Charlie Goodrich of NASA Kennedy Space Center. It features a large space of possible states and 256 different observations. This model was tracked in runs of 618 steps.

The in-situ propellant production model, *ISPP*, has 59 components and represents a chemical processor designed to produce rocket fuel from the Martian atmosphere. Figure 6.10 illustrates the ISPP hardware, though the components in the upper right quadrant have not yet been modeled in Livingstone. The Martian atmosphere would enter the system's sorption pump at the top left. At Kennedy Space Center, a substitute for the Martian atmosphere consisting largely of carbon dioxide, ( $\text{CO}_2$ ), was used to test the apparatus. The  $\text{CO}_2$  is combined with hydrogen ( $\text{H}_2$ ) to produce methane ( $\text{CH}_4$ ) and liquid oxygen ( $\text{O}_2$ ). The methane and oxygen can then be burned by vehicles returning from Mars to Earth. The ISPP model was chosen over other, larger models because failures involving gas flows in the system require far more time for diagnosis under *Livingstone* than any other scenario we have encountered for any model. The ISPP system is designed to be run during the Martian day and shut down at night. This model was tracked through scenarios of 33 steps, approximating one day's worth of commands. Additional runs were made on both the CB and ISPP models. Our results suggest the following.

- *Model growth per time step is small.*

The ISPP model begins at 2933 clauses. This is the number of clauses needed to represent the ISPP system at a single time step, including the full state model  $\mathcal{M}_\Sigma$ . At each time step created by commanding the ISPP system, the transition system structure grows an average of 36 clauses per time step. This small growth is due to the optimizations developed in Chapter 5. Only the component that is currently being commanded and components whose failure could effect it are explicitly modeled at each time by inclusion of a small portion of the transition model,  $\mathcal{M}_\mathcal{T}$ . Similarly, the CB begins at 1126 clauses and grows by an average of 44 clauses per time step.

- *Tracking time steps where no failure occurs takes a very small amount of CPU time.*

Consider Figure 6.11. It illustrates execution time for a CB scenario. The horizontal axis of the graph represents the time steps in the scenario, one time step per each of the 624 commands issued in the scenario. The vertical axis reports the amount of time required by  $L2$  at that step in order to continue tracking trajectories. The odd appearance of the graph is explained by the fact that each step of the scenario requires less than the available clock resolution of 0.016 seconds. Thus the time reported bounces between 0 seconds and 0.016 seconds.

In the first 16 steps, a subset of the circuit breakers are turned off then turned on again. This sequence is then repeated 38 additional times. On the final cycle, one of the circuit breakers fails. Since there are no failures until the final time step, only one trajectory is tracked in this example. Our techniques for adding an additional time step onto an existing trajectory data structure and testing the result for consistency were designed to perform very little work at each time step. Figure 6.11 shows negligible growth in execution time to track a consistent trajectory as the trajectory's length grows. The consistency checking problem is linear in the number of clauses in the model, which is growing very slowly compared to the initial model size.

Figure 6.12 illustrates a 33 step scenario for the ISPP model. The horizontal x-axis of this graph represents the time steps in the scenario. The right vertical axis reports the amount of time required by  $L2$  at that step in order to continue tracking trajectories, denoted by the solid line. The left vertical axis is the number of trajectories being tracked at each point in the scenario, as denoted by the triangles. A failure that occurs at time step 27 cannot be uniquely diagnosed, so multiple trajectories are tracked.

During time steps 0 through 26, no failures occur. During this period, only the nominal trajectory is tracked, and again there is no sign of computational growth per time step. On the 27<sup>th</sup> step, a very tricky flow-related failure becomes visible. The result is eight consistent diagnoses for this failure that are most likely. Between steps 27 and 32, these eight trajectories are extended by nominal transitions and remain consistent. We see an increase in computation per time step versus steps 0 through 26 as there are multiple now

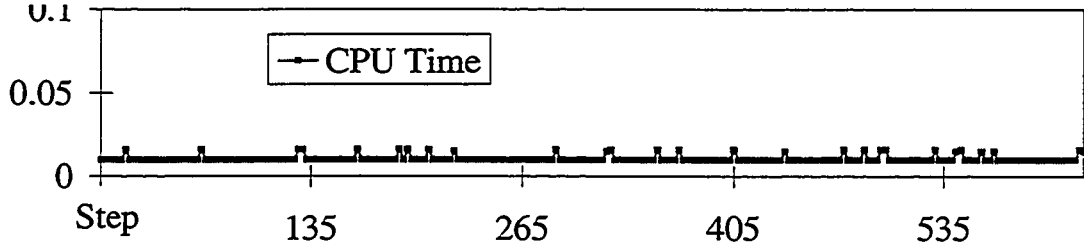


Figure 6.11: CB - Single Failure After 598 Steps

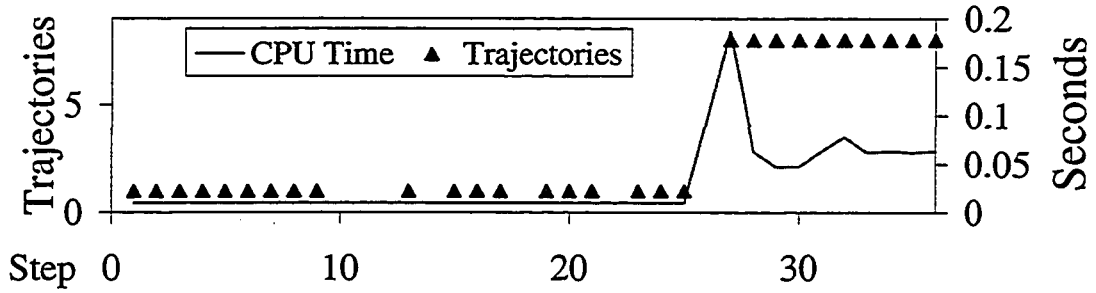


Figure 6.12: ISPP - Independent failures at steps 27, 32 33

trajectories to be extended and checked for consistency. However there is not a significant increase in time as we track the 8 trajectories until additional failures begin to occur at time step 32.

- *Keeping a history does not induce an unreasonable cost when diagnosing a single failure.*

The CPU time for a single CB failure scenario is below clock resolution whether 15 or nearly 600 nominal steps, as in Figure 6.11, precede the failure. This is expected. Consider that when we are tracking the nominal trajectory and it is first made inconsistent by a failure, this inconsistency introduces a single long conflict. The maximum cost of a covering of the conflict set,  $\gamma$ , is initially 1. Intuitively, each diagnosis selects a replacement for one nominal transition in the conflict. We then test the diagnosis for consistency using a linear time procedure. We therefore expect a worst case linear increase in cost for diagnosing a single failure as we increase the length of the trajectory.

- *Because of the accumulation of conflicts, tracking the system through  $k$  failures spread over time can be an easier problem than diagnosing a single failure of cardinality  $k$ .*

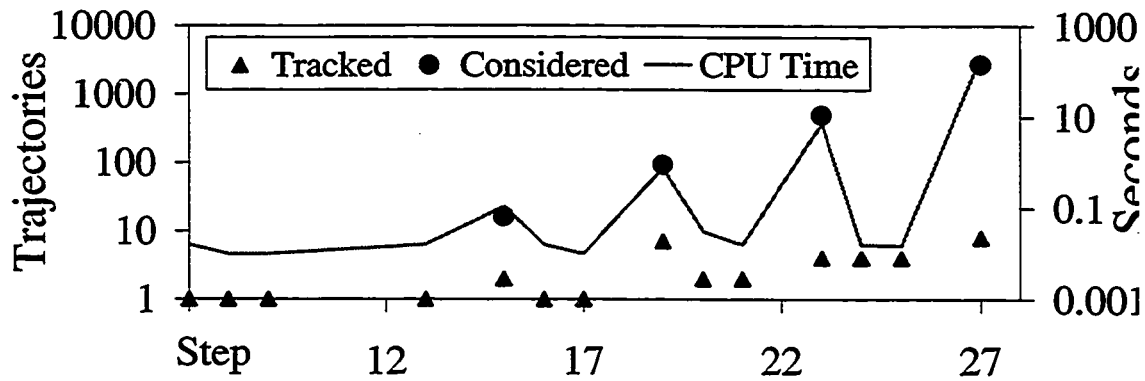


Figure 6.13: ISPP - 4 Identical failures over 27 Steps

Consider the run of Figure 6.12. On step 27, the flow failure occurs. It requires 0.19 seconds to find that there are eight equally likely consistent diagnoses. Eight trajectories result and are tracked until step 32. On step 32, a simpler, unrelated failure occurs, and none of the 8 trajectories is consistent when extended by the nominal transition. Note that *L2* must now re-diagnose the entire history of the system including the flow failure. It does so in just 0.08 seconds, less than half of the time required to diagnose the flow failure alone. The key to this behavior is the conflicts. On step 27, the nominal trajectory is ruled out and *ConflictDB* contains a single conflict. *GenerateCover* returns 28 candidate trajectories, 20 of which are ruled out, adding another 15 candidates to *ConflictDB*. Intuitively, these additional conflicts memoize the result of finding and eliminating the 20 inconsistent candidate trajectories, and any extensions of them. If we call *GenerateCover* with the same  $\gamma$  on the expanded conflict set, it returns only the 8 consistent candidates and requires almost no CPU time. On step 32, the 8 diagnoses are ruled out by conflicts resulting from the simple failure. Since the conflicts from the simple failure involve none of the variables from the flow failure conflicts, the problem decomposes into two subproblems, one of which has previously been solved and memoized by the conflicts.

- Unfortunately, tracking  $k$  related failures over time can also be as computationally intensive as diagnosing a cardinality  $k$  failure.

Figure 6.13 illustrates a sequence of failures for the ISPP system where the conflict coverage problem does not decompose. On step 15 the flow failure is introduced. Repair

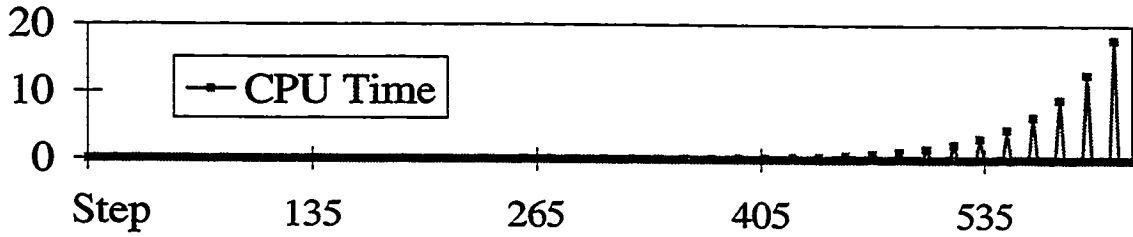


Figure 6.14: CB - 39 Identical failures over 618 Steps

actions are taken and the failure is immediately reintroduced, until a total of four identical failures have occurred. Note that the time axis is logarithmic and the CPU time rises exponentially with the number of failures.

The issue here is that we have thrown out all intervening observations between the first and second occurrences of the failure in order to minimize our model size. Thus when we receive the observations indicating the second occurrence of the same failure, it is consistent to believe that the first failure simply persisted. Conflicts for the later failure are supersets of conflicts introduced by the earlier instance of the failure. Thus, a covering of the conflicts generated for the first failure at  $\gamma = 1$  is also a covering for the conflicts generated at the second failure where  $\gamma = 2$ . Thus there is a slack of 1 in  $\gamma$  that can be used to cover additional conflicts. However, since all conflicts are already covered, there are no constraints on where the additional  $\gamma$  is spent and all possibilities are considered as possible failures. Intuitively, there are not enough conflicts and the algorithm considers multi-ways failures at many time steps. At the fourth failure, 2694 candidates are returned in 174 seconds. An additional 33 seconds are spent determining all but 8 of them are inconsistent. Figure 6.14 illustrates a run wherein the device in the CB model fails on every 16 step cycle and is reset. The device fails a total of 39 times. Again, this clearly shows the exponential growth of tracking time as the number of failures involving the same device grows.

Fortunately, this behavior is largely an artifact of how *CoverTrack* was implemented. It first generates the set of possibly consistent diagnoses by finding a covering for the set of



conflicts that have been discovered during diagnosis thus far. Each of these possibly consistent diagnoses differs from each conflict, and thus does not contain a known conflict. Each possibly consistent diagnosis is then checked for consistency, which potentially reveals additional conflicts. Intuitively, *CoverTrack* is generating many variations of the same possibly consistent diagnosis that in fact all share an inconsistent set of assignments. This occurs only because we generate all diagnoses that cover the current conflict set before testing if any of them are consistent. We avoided this issue when applying *L2* by simply using *CBFS-track* or using a short history with *CoverTrack*. In the conclusion of the thesis we present a simple solution to this problem as an item for future work.

## Chapter 7

# New Approaches to Conformant Planning

### 7.1 Introduction

With complex systems such as spacecraft, we are often faced with situations in which we need to achieve goals even though there are faults present. Unfortunately, these systems are only partially observable, or observations may be costly. For example, sensors dedicated to measuring the internal state of spacecraft are usually quite minimal due to power and weight constraints. Thus, if we diagnose a fault aboard a spacecraft, we might do no better than finding a small set of failures that are equally likely given the limited set of observations.

The preceding chapters explore methods for determining a partial distribution over the possible state an apparatus occupies. Given that the control system no longer knows with certainty what state the apparatus occupies, the question arises of how should one choose actions. An action that achieves some desirable goal when the apparatus is in one possible state may be ineffective or precipitate a disaster if the apparatus is in fact in another state. Our problem is that we need to generate plans that achieve goals even though the exact failure that occurred, and thus the exact state of the spacecraft, cannot be determined. For example, consider the schematic of the Cassini spacecraft propulsion system shown in Figure 7.1. This schematic shows the major tanks and valves of the system, and for clarity omits some parts such as pressure regulators. The purpose of the system is to produce thrust by burning a combination of liquid oxygen (LOX) and hydrogen ( $H_2$ ) in either engine, M1 or M2, shown at the left of the diagram. The system operates as follows: The high pressure helium tank to the right of the diagram contains helium at high pressure. When either of

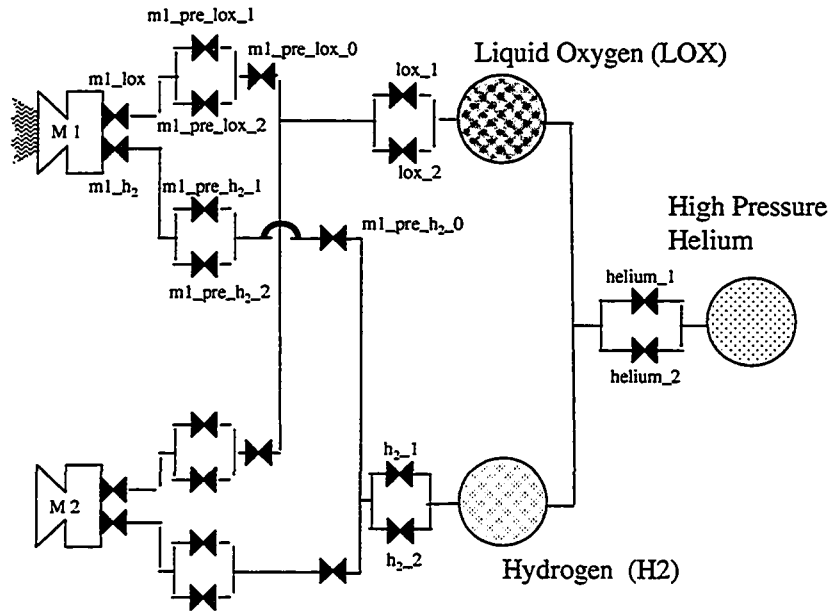


Figure 7.1: Cassini Propulsion System Schematic

the valves `helium_1` or `helium_2` is opened, the liquid oxygen and hydrogen tanks are pressurized. To provide LOX to engine M1, we must open one of `lox_1` or `lox_2`, plus `m1_pre_lox_0`, plus one of `m1_pre_lox_1` or `m1_pre_lox_2`, plus `m1_lox`. To provide  $H_2$  to M1, we must similarly open a path from the hydrogen tank to the valve `m1_h2`. Once a flow of  $H_2$  and LOX to M1 has been enabled, M1 must be ignited. Similarly, we may choose to ignite M2, or ignite M1 and M2 simultaneously. Suppose our goal is to produce thrust when one of either `m1_pre_lox_1` or `m1_pre_lox_2` is stuck closed. There are many plans that produce thrust regardless of which of these conditions holds. Intuitively, we may simply use engine M2, which avoids both of the valves in question. We may also produce a plan that attempts to open both `m1_pre_lox_1` and `m1_pre_lox_2`. Regardless of which valve is failed, LOX will flow to M1 through the other valve.

The Cassini scenario described is an example of a *conformant planning problem*. Conformant planning is a generalization of deterministic planning wherein the task is to generate a plan that moves a system from any one of a number of possible initial states to a state that satisfies a set of goal predicates. In addition, actions may have uncertain outcomes and sensing actions are not available to the plan. We denote the start states and goal constraints

of a planning problem as follows.

- Let  $\mathcal{V}$  be the propositional variables describing the state space of the problem.
- Let  $\mathcal{S}$  be a set of possible initial states. Each member of  $\mathcal{S}$  is an assignment to  $\mathcal{V}$ .
- Let  $\mathcal{G}$  be a set of predicates  $\{g_0 \dots g_n\}$  on assignments to  $\mathcal{V}$ .  $\mathcal{G}$  is the set of goal constraints.

**Definition 7** Given a plan  $p$  and an initial state  $s$ , we define the predicate  $\mathcal{G}_p(s)$  to be true if deterministic execution of  $p$  starting in state  $s$  results in a state that satisfies all  $g_i \in \mathcal{G}$ .

**Definition 8** Given the set of possible initial states  $\mathcal{S}$ , a plan  $p$  is **conformant** with respect to  $\mathcal{S}$  if execution of  $p$  from any of the initial states reaches a state that satisfies  $\mathcal{G}$ . That is, for all  $s \in \mathcal{S}$   $\mathcal{G}_p(s)$ .

That is, the task of conformant planning is to find a plan  $p$  that when executed reaches the a state that satisfies all goals no matter which possible initial state of the system is the actual initial state. The computational challenge of conformant planning lies in the fact that the effects of a plan when executed in one state may be different and highly undesirable when the plan is executed in a different state. Thus one cannot choose an action based on its desired effect given one possible initial state of the system (called a *world* in the conformant planning literature) without in some way considering its unintended effects when it is executed in all other possible initial states. The traditional approach to conformant planning has been to consider the effects of each action under consideration across all worlds simultaneously. Techniques for representing the effect of an action across all worlds include creating a Graphplan-style planning graph (Blum & Furst 1995) for each world and adding constraints between them (Smith & Weld 1998) and binary decision diagrams (Cimatti & Roveri 1999).

In this work, we take a different approach to conformant planning in that we attempt find a plan that works in a single world and extend it to work in all worlds. To do this, we plan in a single world at a time using a deterministic planner. We use the plan generated in each world to influence how plans are generated in the remaining worlds, in order guide the planner toward producing a plan that works in all worlds. Intuitively, if we have  $n$  worlds

that require a domain model consisting of  $m$  variables, then reasoning about all worlds simultaneously presents a planning problem of size  $nm$ . Conformant planning is PSPACE-complete in general, as is deterministic planning (Haslum & Jonsson 1999; Bylander 1991). It remains NP-complete even when we consider only a fixed horizon over which actions may be taken. The worst-case computational cost of propositional planning, known as the plan existence problem, for a single problem of size  $mn$  is  $O(e^{e^{mn}})$  in general and  $O(e^{mn})$  in the fixed horizon case. Alternatively, we propose to make  $l$  attempts at solving problems of size  $m$  and then combine the results, at a cost of  $O(e^{e^m l})$  or  $O(e^{ml})$  for a fixed horizon. Even if  $l > n$ , there is a potential for significant savings.

Conceptually, we can add two kinds of information to the theory in order to increase the likelihood that while planning for world  $\omega$  the planner will return a plan that succeeds in additional worlds. First, we could add information about plans that succeeded in worlds other than  $\omega$  in the hope of generating similar plans. Second, we could add information about plans that failed in other worlds in the hope of avoiding similar plans. We have explored both of these methods. In *fragment-based* conformant planning, we generate a plan for  $\omega$ . We then assert the plan for  $\omega$ , referred to as a *fragment*, into the theory. In effect, when we plan for the next world, the planner will be forced to include the actions known to safely achieve the goal in world  $\omega$ . In *conflict-based* conformant planning, we check the execution of a plan generated in one world in all other worlds. If the plan is not safe and conformant, we use information about how it failed in order to rule out similar plans from being generated. Intuitively, in conflict-based conformant planning each world is informed as to the space of plans the other worlds cannot accept. In fragment-based planning, each world is given a space of plans that other worlds can accept. These two approaches to conformant planning are described below.

## 7.2 A Fragment-based Conformant Planner

To illustrate our approach, let us consider the extremely simple conformant planning domain<sup>1</sup> shown in Figure 7.2. Here we have a pressure source followed by six valves, v1 through v6. Assume that we have commanded all six valves to open, and yet no flow is

<sup>1</sup>This domain is isomorphic to the widely-used bomb in the toilet domain (McDermott 1987)

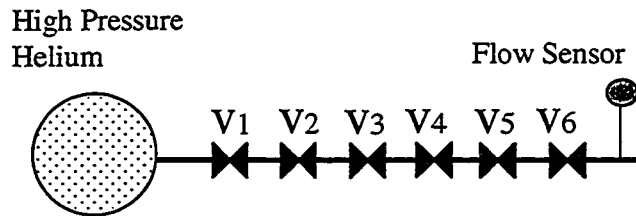


Figure 7.2: A Simplistic Valve System

Time	Action
1	open v1
2	recharge
3	open v2
4	recharge
5	open v3
6	recharge
7	open v4
8	recharge
9	open v5
10	recharge
11	open v6

Figure 7.3: A Plan For 6 Valves in A Row

detected out of the pipe. For simplicity in the explication, we will assume that we know that at most one valve has failed to open when commanded and that commanding the valve open again is the remedy. To keep the problem from becoming trivial, we will assume that we don't have enough power to simultaneously open multiple valves. We will model this with an action called *recharge* that is required after any valve is opened.

Figure 8.1 illustrates a successful plan for this problem requiring 11 time steps, the minimal number. Note that given 11 time steps there are  $7^{11}$  possible action sequences and  $6!$  conformant plans.

Note also that the first action of this conformant plan is itself a plan to achieve the goal in the world in the valve which did not open was v1. Similarly, the conformant plan contains a one-step plan for every other world. It also contains actions, in this case re-charging, that reconcile the post-conditions of the plan for each world with the preconditions for another world. These characteristics are not specific to this particular conformant plan or even this planning domain. By definition, if a conformant plan exists for a goal and a set

Column 1 Plan for $w_1$	Column 2 Fragments for $w_2$	Column 3 Plan for $\{w_1, w_2\}$	Column 4 Extracted Fragment	Column 5 Fragments for $w_3$	Column 6 Plan for $\{w_1, w_2, w_3\}$
1 open v1	1 open v1	1 open v1	1	1 open v1	1 open v1
2	2	2	2	2	2 recharge
3	3	3 recharge	3	3	3 open v3
4	4	4	4	4	4 recharge
5	5	5 open v2	5 open v2	5 open v2	5 open v2

Figure 7.4: Generating A Plan for Restoring Flow When One of Three Valves is Closed

of worlds, then the conformant plan must contain a set of actions that achieve the goal in each individual world. This property can be made more useful if stated slightly differently. If a conformant plan exists for a set of worlds, there must exist some plan for each world that can be augmented to form a conformant plan. In particular, we must add actions that achieve the goal in other worlds and ensure the pre- and post- conditions of all actions are satisfied. This insight forms the basis of fragment-based conformant planning.

In fragment-based conformant planning, we attempt to create a plan that achieves the goal in all worlds by progressively assembling plan fragments that achieve the goal in each world. The approach is based upon the observation that, by definition, a conformant plan achieves the goal in all worlds and thus, for each world, must contain a set of actions that achieves the goal. Intuitively, we can construct a conformant plan by finding the appropriate fragment for each world and assembling them. We will do this by finding a plan for an initial world  $w_i$  and asserting this plan, referred to as a *fragment*, into the domain theory. During subsequent planning, the planner is thus forced to include this fragment in all subsequent plans. We then plan for the next world,  $w_j$ . In effect, we are attempting to extend the initial fragment into a plan that is conformant for both  $w_i$  and  $w_j$ . If we are successful, we again attempt to add an additional world. If we can find a plan for the final world that includes the fragments found for all other worlds, we have a conformant plan. Consider the following example where we only have three valves.

**Example 9** We have three valves, v1, v2 and v3, that might not be open. To ensure flow, each must be opened. Figure 7.4 illustrates one possible assembly of fragments for this example. In the figure, we refer to the world in which valve  $i$  was the valve that failed to

open as world  $w_i$ . Note in this example we choose to use a planning style in which plan actions are assigned to fixed points in time, as is done in Graphplan and SATPlan (Kautz & Selman 1996) style planners. This is not a requirement for fragment-based planning.

#### Column 1

In world  $w_1$ , the closed valve is  $v1$ . We first create a plan for  $w_1$ , which is simply to open  $v1$ .

#### Column 2

We then use this plan for  $w_1$  as a fragment of the conformant plan that must appear in all subsequent plans. This determines the initial conditions for planning for the second world,  $w_2$ , where  $v2$  has failed to open.

#### Column 3

We then plan for the situation in which the  $v2$  is still closed and the first action of the plan must be to open  $v1$ . This results in a plan that succeeds in  $w_2$ . We then check that the plan still achieves the goal in  $w_1$ . In Column 3, we now have a plan that achieves the goal in all worlds considered thus far. In the next column, we extract a fragment for  $w_2$  from this plan.

#### Column 4

Conceptually, we can think of the plan in Column 3 as consisting of the fragments that were required for previous worlds (here, *open v1*), a fragment required to achieve the goal in  $w_2$  (here, *open v2*), and a set of repair actions that allow these fragments to coexist (here *recharge*). In fact, there are 11 sets of repair actions that allow the chosen fragments for  $w_1$  and  $w_2$  to coexist<sup>2</sup>. Only four of these allow us to later add a fragment for  $w_3$ .

In order to avoid constraining all future plans to the extent possible, we would like to avoid asserting into our fragment set any repair actions that can easily be re-derived later. Currently we use a simple procedure to remove repair actions from a plan. We first remove the fragments from previous worlds. They will be added back in in the next step.

---

<sup>2</sup>If we use only recharges, we can fit 8 variations in the three available time steps. To include a open, time 2 and time 4 must contain recharges while time 3 can open of any of 3 valves.



We then perform a search that, given  $w_2$ , removes any action whose post-conditions are true before it is executed. This generally removes repair actions. If all repair actions are not removed, we may simply encounter additional backtracking. In this example, the plan is reduced to *open v2*.

#### Column 5

The fragment extracted from the plan for  $w_2$  is asserted into the planning domain along with the fragment for  $w_1$ . Subsequent plans will be required to include these fragments.

#### Column 6

Finally, we plan for the situation in which the closed valve is  $v_3$ , the first action must be to open  $v_1$  and the last action must be to open  $v_2$ . We find a fragment that achieves the goal given the closed valve is  $v_3$  and choose a set of repair actions that fit it within the fragments previously asserted for  $w_1$  and  $w_2$ . We now have a conformant plan.

Note that while every conformant plan has a fragment for each world, not every fragment for a world can be extended into a conformant plan. The chosen fragment for achieving the goal in one world may conflict with every fragment for achieving the goal in a world that has not yet been considered. Figure 7.5 illustrates a simple and incomplete fragment-based planner to serve as a straw man. It produces plans in the manner illustrated in Figure 7.4. We begin with a domain model in *Domain*, the set of initial worlds in *Remain*, and *Done* equal to the empty set. We select a world  $\omega$ , and use a deterministic planner to find a plan that succeeds in  $\omega$ . If no such plan is found, or if the plan does not succeed in the worlds we have previously considered, the procedure gives up. If a suitable plan is found, it is reduced to the actions needed to achieve the goal in world  $\omega$ . The variable *frag* then contains a plan that will achieve the goal in world  $\omega$ . We must now find a fragment for the next world given the fragments we have found for the worlds selected thus far. This is accomplished with a recursive call that adds the current fragment to the planning domain. This forces the next call to the planner to find a plan that integrates achievement of the goal in the next world selected with the pre- or post- conditions of the fragments that achieve the goal within the previously explored worlds.

As stated, the algorithm in Figure 7.5 does not do any backtracking. Unfortunately,

```

proc simpleFragPlan(Domain, Remain, Done) {
  select  $\omega$  from Remain
  Plan=plan(Domain, $\omega$ )
  if(no Plan)
    return Failed
  if(Plan does not achieve goal in Done)
    return Failed
  if(Remain -  $\omega$  =  $\emptyset$ )
    return Plan
  frag=ExtractFragment(Plan, $\omega$ )
  simpleFragPlan(Domain+frag, Remain -  $\omega$ , Done+ $\omega$ )
}

```

Figure 7.5: Simple Fragment Planner

not every plan for a single world  $\omega$  can be extended into a conformant plan over a set of worlds that includes  $\omega$ . There are two ways that a set of fragments might fail to extend into a conformant plan. First, it may be the case that the fragment we arbitrarily choose for the current world,  $w_i$ , may be incompatible with all fragments for achieving the goal in some subsequent world  $w_j$ . Second, when we extend the fragment for  $w_i$  to achieve the goal under  $w_j$  by choosing additional actions, there is no guarantee the extended plan still accomplishes the goal when executed in world  $w_i$ . We must check the augmented plan in  $w_i$  to ensure it maintains this property. For either one of these failures, we will be forced to undo some subset of the existing choices before any further progress can be made.

Figure 7.6 illustrates a complete, backtracking fragment-based planner. Intuitively, a suitable fragment for a conformant plan is one that results in a plan that satisfies the goal in the new world  $\omega$  plus all worlds considered thus far (the set *Done*), and allows a suitable fragment to be found for the remaining worlds (the set *Remain*). We initially call the planner with *Domain* equal to the planning domain, *Remain* equal to the initial world set, and *Done* empty. As the algorithm recurs, an additional world  $\omega$  is moved from *Remain* to *Done*, and the fragment that satisfies the goal in world  $\omega$  and does not disrupt the existing fragments for *Done* is added to *Domain*. We use the non-deterministic operator **Choose** to ensure we eventually consider all fragments for a given world. This is a backtrack point if the chosen fragment does not work out. Consider the operation of the algorithm after some number of worlds have been added to *Done* and the corresponding fragments

```

proc completePlan(Domain, Remain, Done) {
  select  $\omega$  from Remain
  Choose Plan from Plan(Domain,  $\omega$ )
  if(no Plan found)
    Fail
  if(Plan satisfies all worlds in Done) {
    if(Remain= $\emptyset$ ) {
      report Plan as conformant plan
      return true
    }
    frag=ExtractFragment(Plan, $\omega$ );
    if(completePlan(Domain+frag, Remain- $\omega$ , Done+ $\omega$ ))
      return true
  }
}

```

Figure 7.6: A Recursive, Complete Fragment Planner

added to *Domain*. If no plan can be found for the next world  $\omega$ , we must backtrack to the fragment choice point at the previous recursion level. If a plan is found and no worlds remain, we have found a conformant plan. Otherwise, we extract from the plan a fragment that represents the actions needed to achieve the goal in  $\omega$  and add the fragment to *Domain*. We then attempt to complete the conformant plan for the remaining worlds. This procedure is complete.

Note that the world ordering chosen by *Select* and the order fragments are chosen by *Choose* have no impact on completeness in the procedure of Figure 7.6. However, they may have a significant impact upon the amount of backtracking that is performed. Numerous search strategies are possible. A simple strategy is to perform complete backtracking on the possible fragments of each world before backtracking to the previous world. Alternatively, we can use stochastic sampling of the fragments for each world. In either case, the order in which we select worlds is also a consideration, since some worlds may be tougher than others. In the next section, we discuss strategies for these choices in the context of this complete planning algorithm and an incomplete, randomized variation. Note also that we assume the Plan procedure can be forced to include the fragment actions in the plan it finds for the current world. Options for implementing Plan are discussed in the following section.

### 7.3 Search Strategies for Fragment-based Planning

In this section, we examine search procedures for fragment-based planning. The complete planning algorithm of Figure 7.6 uses chronological backtracking. We also consider incomplete, randomized search strategies, given their effectiveness in comparison to systematic algorithms in many domains (Gomes *et al.* 1998; Selman, Kautz, & Cohen 1996). Intuitively, we can think of planning with fragments as analogous to a constraint satisfaction problem. We have a set of variables (worlds) for which we must choose assignments (fragments) so as to satisfy a set of constraints (the conformant planning domain and goal). Decisions about how variables are ordered and which assignments are chosen will directly impact how much backtracking will be required to retract infeasible variable assignments and how quickly a solution will be found. Therefore, each of our search strategies will have to specify the following characteristics.

- *Variable Ordering:* In what order will the search consider the worlds?
- *Frustration Level:* How many unsuccessful fragment combinations will the search consider before backtracking to a previously considered world?
- *Backtracking Distance:* When the search decides to backtrack, how many fragment choice points will it backtrack over?

Note that in the complete case, either systematic or randomized, the answers to these questions have no impact upon completeness. All fragment sets will eventually be attempted, and each conformant plan is one of those sets, so ordering choices impact efficiency only. With an incomplete planner, we are explicitly assuming only a portion of the possible fragment sets will be considered. The order in which we consider the worlds and previous fragment choices will have a significant impact upon which subsequent fragments are considered. The goal is to develop search strategies that consider fragment sets in which a conformant plan is likely to be found.

Figure 7.7 illustrates a fragment-based planner with the flexibility to accommodate a variety of search strategies, both complete and incomplete, systematic or randomized. It differs from the complete, systematic planner of Figure 7.6 in several respects. First, since the search procedure will be examining and modifying the world ordering, we store it in

```

proc fragPlan(Domain, Worlds) {
  select  $\omega$  from Worlds
  worldStack=empty
  loop{
    Plan=Plan(Domain, $\omega$ )
    if(Plan  $\neq \emptyset$  & Plan satisfies all worlds on worldStack) {
      if(Worlds  $\subseteq$  worldStack)
        return Plan
      newFrag=ExtractFragment(Plan, $\omega$ )
      Domain= Domain + newFrag
      push( $\omega$ ,worldStack)
      select  $\omega$  from Worlds
    }
    if (Plan= $\emptyset$  or Frustrated() ) {
      // For each world removed from the stack, we must remove
      // the corresponding fragment from Domain
      stack = adjustStack(worldStack,Domain,failures)
      select  $\omega$  from Worlds
    }
  }
}

```

Figure 7.7: Flexible Fragment Planner

an explicit stack rather than through recursion. Second, the search need not consider all fragments for a new world given a set of worlds and fragments. Instead, the search gives up on the current fragment choices and world selection whenever it becomes frustrated with the number of fragments it has generated for the current world without finding one consistent with fragments it previously chose. Third, when the search gives up, it may undo as many previous world selections as desired and continue the search from there. Finally, the *Plan* subroutine need not be complete and may employ randomized search procedures. This will be critical to the effectiveness of our search procedures. We have investigated several search procedures that make different choices for variable ordering, frustration level and backtracking distance. Of these, chronological backtracking and two randomized searches that provided interesting experimental results are described below.

### Chronological Backtracking

In chronological backtracking, upon failure we undo the last choice made and replace it with its successor until all choices have been considered. To implement backtracking,

Time	Action
1	open v5
2	flush
3	open v4
4	flush
5	open v3
6	flush
7	
8	open v2
9	flush
10	
11	open v1

Figure 7.8: A Simple Fragment-based Plan That Failed

*Plan* must be implemented such that successive calls with the same *Domain* and  $\omega$  return successive plans according to some ordering, and *Frustrated* must always be false. Thus, *Plan* will return plans given the current fragments until all such plans have been exhausted. The *adjustStack* and *Select* procedure must then remove the last world from the stack, along with its current choice of fragment, and install it as the current world  $\omega$ . We then resume generating possible plans for the fragments associated with the worlds remaining on the stack. The disadvantage of this strategy is that it considers every possible choice for the current fragment before considering the previous fragment choice. As a result, the number of fragments considered will be exponential in the number of choices made between the choice that must be changed to enable a plan and detection that the planning attempt has failed. Consider the situation of Figure 8.22. Suppose the first fragment chosen was the action *open v2* at time step 8. No conformant plan can result from extending this fragment. However the planner does not fail until it attempts to place the open action for the sixth valve, as illustrated in the figure. The planner must then systematically consider all placements of fragments around the *open v2* action before reconsidering that action itself.

#### Stochastic Probing (Langley 1992)

In this context, a probe consists of one selection of the world order and a choice of fragment for each world. To implement stochastic probing, we select a world at random and use a randomized planner to find one possible fragment. We subsequently select worlds

randomly from the set of unconsidered worlds, and find fragments that are consistent with the existing fragment set. If we reach a point where we cannot find the next fragment for the world ordering, we throw out all fragments and the world sequence and begin again. To implement this search strategy, *Select* must randomly select a world that's not on the stack, *Plan* must be a randomized planner, *Frustrated* must always be true, and *adjustStack* must completely empty the world stack and *Domain*. We expect this strategy to work well in situations similar to Figure 8.22 where the problematic choice was made far back in the stack of decisions, but is not detectable until much later. Note that this strategy is randomized but can maintain completeness if we prevent *Plan* from generating the same plan twice. This can be accomplished through the use of nogoods. We discuss this further in the section on the implementation of the *Plan* procedure.

### Bubbling

Bubbling refers to the motion of worlds for which the search is having difficulty finding a plan toward the bottom of the stack. In bubbling, *Plan* is randomized and *Frustrated* becomes true after some small, fixed number of planning attempts. When the planner becomes frustrated in its attempts to find a plan for  $\omega$  given all of the worlds, and implicitly fragments, on the stack, *adjustStack* pops the last world off the stack. The variable  $\omega$  is left with its selected value. Thus the search continues for a fragment for  $\omega$ , but within the context of a smaller set of worlds and associated fragments. This continues until we find a plan that satisfies  $\omega$  and the stacked worlds, or until the stack consists only of  $\omega$  and we find a plan for it. Intuitively, the problematic value for  $\omega$  bubbles up the stack until a fragment is found. The search then returns to finding fragments for the worlds not on the stack. We can refine the variable ordering strategy of bubbling to prefer that heavily constrained worlds are solved first, in a manner analogous to squeaky-wheel optimization (Joslin & Clements 1999). We approximate this by introducing a notion of difficulty. Each time *Plan* fails to find a fragment, the difficulty of each world on the stack is incremented. After  $\omega$  has been satisfied, we can select the next world to attempt based upon this estimate of its difficulty. We expect this search to do well in domains where a small subset of the worlds are significantly more difficult to satisfy than the remaining worlds.

## 7.4 Implementation Using a SAT Planner

The fragment-based approach does not require a specific planning approach be used to implement the Plan procedure that is used on individual worlds. It only requires that we are able to force the Plan procedure to include actions for previous fragments in its plan for the current world. This can be easily accomplished for many planning styles. Partial-order planners begin with an empty plan and add actions that link the initial state to the goal state and remove conflicts between actions. We can enforce the inclusion of fragments by initializing the procedure with a non-empty initial plan consisting of the fragments. In Graphplan-style planners, the backtracking search selects actions that lead from the goal back to the initial conditions. At each level of this search, we can simply force the search to include the appropriate fragment actions in its set of actions. SAT-plan based procedures build a propositional representation of the possible plans. We can simply assert that the proposition corresponding to selecting each fragment action must be true.

To implement our Plan procedure, we chose to work within the framework of planning as propositional satisfiability and chose Blackbox(Kautz & Selman 1999) as its basis. Blackbox compiles a planning domain into a plan graph(Blum & Furst 1995) then converts the plan graph into a propositional formula that describes the planning problem. For each action and time step, the formula contains a propositional variable  $A$  that, if true, corresponds to taking that action at the given time. It is consistent for  $A$  to be true if and only if the propositional variables representing the preconditions and postconditions of action  $A$  are true as well. Thus, a satisfying assignment to the variables of the formula corresponds to a plan. Blackbox then calls a SAT procedure to find a satisfying assignment for the propositional formula. In order to represent uncertainty in the initial state of the world, we add to the propositional formula a set of propositional variables, each of which represents the possibility that a particular world is the actual world. We modify the propositional formula such that if a world variable is true, then the propositional variables that represent the conditions of that world must be true as well. Thus for any world, we can assert that the variable corresponding to the world is true, and Blackbox will find a plan that achieves the goals in that world. Note that we cannot create a conformant plan by specifying the initial



world as a disjunction of the corresponding world variables as a SAT solver will simply pick the most convenient world. As a result, we would only be guaranteed that the resulting plan would satisfy the goal in at least one of the possible worlds. We seek a plan that safely reaches the goal in every world.

Representing the planning domain as a propositional formula and planning in a single world as propositional satisfiability has numerous advantages. First, we can employ fast, randomized propositional satisfiability engines. We have performed experiments using Chaff (Moskewicz *et al.* 2001), Satz (Li & Anbulagan 1997), and a stripped-down version of Satz, from which we removed most preprocessing of the propositional formula designed to improve performance on large SAT instances. Second, addition of fragments to the planning domain is trivial. We simply assert that the propositional variable corresponding to each action in the fragment must be true. The SAT procedure is then constrained to find only plans wherein those actions are taken. The primary drawback of using the Black-box style of propositional encoding for fragment-based conformant planning involves its Graphplan-style handling of time. The propositional encoding is created from a planning structure with a fixed number of time steps. The planning process assigns actions to specific points in time. Thus, all fragments are fixed at a given time. Since we do not know how we will need to extend the existing fragments, fixing the time points of actions is constraining. Consider the bomb in the toilet plan generated in Figure 8.22. We have considered worlds  $w_1$  through  $w_5$  and have generated a plan that is conformant for those worlds. It can be made conformant for  $w_6$  simply by adding an open and recharge action before any open action. However, the way the planner has laid out the actions for the first five worlds, the two remaining time steps are not adjacent, and the two necessary actions cannot be inserted. The planner will have to backtrack, backjump or restart to remove the open action from time 8. We could potentially address this type of timing issue with an encoding trick in the propositional representation that would allow us to move fragments so as to coalesce or create places to insert actions. However, since the core issue is flexibility in ordering actions, it might be more fruitful to consider mapping fragment-based planning into a partially-ordered planning framework. This is beyond the scope of this work.

## 7.5 A Conflict-based Conformant Planner

Both fragment-based and conflict-based conformant planning plan on a single world at a time and use the results to focus subsequent planning attempts on finding conformant plans. The fragment-based approach forces subsequent planning attempts to include plan fragments known to work in previously considered worlds. The conflict-based approach tests whether a plan generated for a single world achieves the goal in all worlds. If not, it forces subsequent planning attempts to avoid plan fragments that are known not to appear in any conformant plan. The conflict-based approach operates as follows. Given a plan generated for a single world, we can easily test whether the plan achieves the goal in all worlds. If not, we analyze the representation of the planning domain and the plan to find if there is a set of actions in the current plan that must prevent the goal from being achieved. This subset of the plan actions, referred to as a *nogood*, cannot appear in any plan. Note that a sequence of actions that causes the current plan to fail is only a *nogood* if all plans that contain the sequence fail. Addition of the *nogood* to the planning domain rules out a space of non-conformant plans from the planning search space, focusing the planner toward producing conformant plans. Using a propositional encoding makes generation of *nogoods* quite simple. Suppose we have generated a plan that achieves the goal in world  $w_i$ . However, upon testing the plan in  $w_j$ , we find the plan does not achieve the goal. Given the propositional encoding, we can easily determine which subsequence of the plan entails the negation of the goal. Since the negation of the goal is entailed by these actions, no plan that contains this subsequence can achieve the goal in all worlds. We may assert the negation of the *nogood* into the propositional encoding in order to prevent the planner from considering any such plan. The conflict-based conformant planner of Figure 7.9 embodies this strategy.

We assume that  $wff$  is a propositional formula that describes the planning domain given any one initial state. We choose a world  $\omega$  and apply a SAT solver to the conjunction of the initial world assignment and  $wff$ , yielding a plan  $P$ . We then check  $P$  for consistency with the domain representation and each other world, in essence simulating its execution on each world. If  $P$  is consistent with all active goals and safety constraints for each world, then

```

proc conflictPlan(wff, Worlds) {
  for N times {
    /* Generate Plan in one world */
    choose initial world  $\omega$  from Worlds
    Plan=satisfy(wff $\wedge\omega$ )

    /* Check if Plan is safe and useful in all worlds */
    Conformant=true;
    for w in (Worlds- $\omega$ ) {
      noGood=checkConsistency(wff $\wedge w \wedge$ Plan)
      if (noGood) {
        Conformant=false
        /* Rule out similar solutions in SAT solver */
        wff =  $\neg$ noGood $\wedge$ wff }
      }
    /* If Plan passed the gauntlet, return it */
    if (Conformant)
      return Plan;
  }
  /* None of N attempts found a safe, conformant plan */
  return NULL;
}

```

Figure 7.9: A Conflict-Based Conformant Planner

$P$  is conformant. If there is an inconsistency, *checkConsistency* traces the inconsistency back through the propositional representation to the sequence of actions that supports it. That sequence of actions is returned as *noGood*. Conceptually, *noGood* prevents a goal from being achieved in at least one world. Thus, no assignment that includes *noGood* is a satisfying assignment. We then add  $\neg noGood$  to *wff*. This has the effect of ruling out all plans that contain *noGood* regardless of the world we are considering during the SAT phase. We then re-invoke the SAT solver on the more constrained problem  $\neg noGood \wedge wff$ . In this way, with each planning attempt the propositional representation of the planning domain becomes more focused upon producing plans that do not violate any constraints in any world.

Note that every non-conformant plan encoded in the SATPLAN-style representation we use in fact contains a *nogood*. A complete assignment to a SAT encoding of a plan specifies both what actions were taken and what actions were not taken in the finite number of available time steps. The complete plan therefore cannot be augmented with additional actions



<pre> recharge()  pre:               eff:  (and charged) </pre>
---

One restriction of planners that support only simple STRIPS actions is that they cannot represent actions with conditional outcomes, that is actions whose effects depend upon the state of the world in which they are applied. Consider our Cassini example. We wish to represent the possibility that some valves are permanently stuck closed and will not open. Thus, the effect of applying the open action to a valve in the Cassini domain is conditional upon whether or not the valve is stuck.

<pre> open(valve)  pre:  (and  charged)                 eff:  (and  (not charged)                         (when  (not (stuck valve))                               (opened valve))) </pre>
--

That is, when we apply the open operator to a valve, if the valve is not currently stuck, it becomes open. In any case, the system is no longer charged. As originally described, Graphplan makes the assumption that operators will not contain such conditional effects. Thus, our additional implementation of the fragment-based conformant planner was unable to handle conditional effects, and an extension was needed. Several authors have described methods that allow Graphplan to handle operators with conditional effects (Gazen and Knoblock 1997, Koehler et al. 1997, Anderson, Smith and Weld 1998). These techniques are useful for deterministic planning, and are not directly applicable to the conformant planning problem. However, our solution is an extension of the approach of Gazen and Knoblock, so we shall describe it first. This approach breaks operators with conditional effects up into a number of separate operators, referred to as *aspects*. Each aspect describes one possible outcome of the operator. Intuitively, we force the planner to choose which effect of the operator will hold in the plan by forcing it to choose which aspect to include in the plan. This is achieved by creating an aspect for each minimal consistent combinations of antecedents in the conditional effects of an operator. To illustrate, consider the following generic operator with one unconditional effect and two conditional effects:

OP	pre: P
	eff: (and E (when C <sub>1</sub> F) (when C <sub>2</sub> G))

This operator would be expanded into the following four STRIPS operators:

OP1	pre: (and P)
	eff: E
OP2	pre: (and P C <sub>1</sub> )
	eff: (and E F)
OP3	pre: (and P C <sub>2</sub> )
	eff: (and E G)
OP4	pre: (and P C <sub>1</sub> C <sub>2</sub> )
	eff: (and E F G)

Intuitively, if the planner requires effects E and G of operator OP, then it must choose to insert operator OP3 into the plan. As with any other action, this will require the planner to ensure that conditions P and C<sub>2</sub> are true when OP3 appears. Unfortunately, this technique can only be used when the start state of the plan is known. Otherwise, we cannot guarantee that condition C<sub>2</sub> will be true when OP3 appears. For example, consider the open operator in the Cassini domain. This can be separated into the following aspects:

open <sub>open</sub> (valve)	pre: (and charged
	(not (stuck valve)))
	eff: (and (opened valve)
	(not charged))
open <sub>noop</sub> (valve)	pre: (and charged
	(stuck valve))
	eff: (and (not charged))

Aspect open<sub>open</sub> applies when the valve is not stuck, and open<sub>noop</sub>, which fundamentally does nothing, applies when the valve is stuck. A conformant plan for the Cassini domain

must potentially apply in an initial state where a valve is stuck, and a second initial state where the valve is not stuck. Thus neither one of these aspects of the open operator correctly captures the operation of the open operator in all worlds. We must have some way of maintaining the uncertainty of operator open as we are generating the conformant plan.

Our solution is to use a simple generalization of the technique of Gazen and Knoblock. If operator OP appears in a plan, then an aspect of OP must apply at the point at which OP appears. Gazen and Knoblock enforce this constraint by forcing the planner to choose a specific aspect that will apply. Intuitively, we encode the constraint itself and allow each world in which the plan must apply to choose a different aspect if necessary. This is accomplished fairly easy given our world at a time approach to conformant planning. Recall that as a fragment is generated for each world, we assert that fragment into the planning domain. This forces plans for subsequent worlds to include the actions that achieve the goal in previously explored worlds. Accommodating conditional actions requires only two basic modifications to the fragment based planning algorithm. First, following Gazen and Knoblock, we replace each conditional operator with a set of aspects. The deterministic planner that provides a fragment for each world is unmodified, but plans with the aspects. Thus, it chooses a specific aspect of each conditional operator that is appropriate to the current world. The second modification concerns how fragments are asserted into the planning domain. When asserting the newly generated fragment into the domain, we do not assert the aspects chosen by the deterministic planner for the current world. Instead, for each aspect  $OP_i$  of operator OP that appears in the fragment, we assert the disjunction of the set of all aspects of OP. Thus planning attempts for subsequent worlds are not constrained to choose the same aspect of OP as applies in the current world. They are constrained to choose some aspect of OP at the point at which OP must appear.

Figure 7.10 gives an intuition of this process using the two aspects of the open operation from the Cassini domain. Imagine our goal is to open one of a set of valves, all but one of which are stuck closed. Let  $w_i$  denote the world where valve  $i$  is not stuck. In Column 1, we plan for  $w_1$ , using the aspects of the open operator. The resulting plan for  $w_1$  is to apply the  $open_{open}$  aspect to v1. Rather than asserting this aspect directly into the domain

Column 1 Plan for $w_1$	Column 2 Fragments for $w_2$	Column 3 Plan for $w_2$	Column 4 Fragments $w_3$
1 $\text{open}_{\text{open}} v1$	1 $\text{open}_{\text{open}} v1 \vee \text{open}_{\text{noop}} v1$	1 $\text{open}_{\text{noop}} v1$	1 $\text{open}_{\text{open}} v1 \vee \text{open}_{\text{noop}} v1$
2	2	2	2
3	3	3 recharge	3
4	4	4	4
5	5	5 $\text{open}_{\text{open}} v2$	5 $\text{open}_{\text{open}} v2 \vee \text{open}_{\text{noop}} v2$

Figure 7.10: Generating A Plan With Aspects

in Column 2, we assert the disjunction of all aspects of open. Intuitively, the plan for  $w_1$  has committed to using the open action, but a different aspect of the operator may be consistent with each world. In column 3, we have asserted  $w_2$  and generated a plan. Assertion of  $w_2$ , where only  $v2$  can be opened, selects an aspect from the disjunction at time step 1. In addition, the planner adds a second aspect to open  $v2$  at time step 5, plus the necessary recharge action at time step 3. Note that if we wished to read out a plan for worlds  $w_1$  and  $w_2$  from this structure, we would substitute the corresponding conditional action for each aspect, resulting in the plan, open  $V1$ ; recharge; open  $V2$ . Instead, we convert the aspect at time step 5 to a disjunction of aspects, assert it into the domain, and continue as with the standard fragment-based planning algorithm illustrated in Figure 7.4.

## 7.7 An Extension to Handle Ramifications

Even given the capability to represent actions with conditional outcomes, many planners from the literature and our Blackbox-based planner cannot directly represent the Cassini problem, illustrated again in Figure 7.11. The problem lies in the practice of representing the ramifications of operators, or their effect on the world, as post-conditions. Intuitively, the use of pre- and post-conditions to capture the effect of operators is useful when the ramifications of an operator are local to a small set of entities that are in some sense local to the operator. For example, in block stacking problems the actions that stack one block upon another effect the block being stacked and the block upon which it is stacked. In a logistics planning problem, moving a truck with a package in it effects the location of the truck



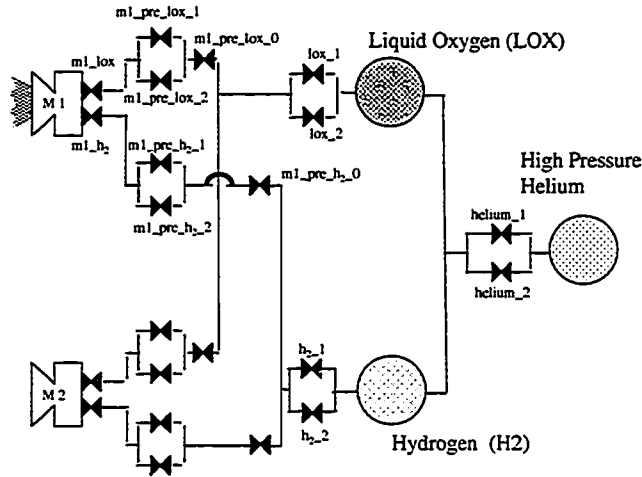


Figure 7.11: Cassini Propulsion System Schematic

and the package. This approach is not scalable when the effect of actions may propagate through many entities, which is precisely the case in the type of machine control problems upon which we would like to focus. Consider the Cassini example and the ramifications of the open operator. Consider first when the operator is applied to the valve `helium_1`. If all other valves are open, then every point in the system is pressurized with either helium, LOX or  $H_2$  as appropriate. If no other valves are open, then the LOX and  $H_2$  tanks and their downstream valves are pressurized. Each of the combinations of valve configurations between these two extremes implies a different set of ramifications for open operator when applied to the valve `helium_1`. In contrast, when applied to the valve `m1_lox`, the open operator can have only two different ramifications (either M1 is receiving LOX or it is not) but which ramification is correct depends upon the configuration of much of the rest of the propulsion system.

There are several possible approaches to this problem. First, we might attempt to expand the pre-conditions of our operators, restricting how they are applied and thus reducing the complexity of capturing their ramifications. For example, if the open operator only applies to valves where pressure is present (initially `helium_1` and `helium_2`), its effect is simply to pressurize the next valves in the path to the engines. This effectively captures

propagation of pressure in the Cassini system from the helium tank to the engines. Unfortunately, this has the effect of ruling out all plans where the valves are not opened in this arbitrary order. This may increase planning complexity, or if there are other constraints on the operators, rule out all valid plans. Second, we might pre-compile the operators of a domain (*e.g.*, valves and pipes) into a set of operators for a problem (*e.g.*, the Cassini propulsion system) that capture the propagation of the properties of interest under any circumstances. For example, given the layout of the Cassini we could generate an operator for opening `helium_1` whose conditional effects capture the pressure propagation given any of the 4 million configurations of the 22 valves. In general, doing this compactly would require having a model of the propagation of interest and embedding it into each operator. Given this model, a third option is to capture only local effects in the operator definitions, then use the model to infer non-local propagation. This option is the one we chose to implement.

Conceptually adding a model of non-local ramifications, which we refer to as the background model, to our Blackbox-based planner is quite simple. Blackbox compiles the planning domain model into a propositional formula. A satisfying assignment to this formula represents a plan. Conceptually, we simply need to augment this formula with additional model fragments that capture propagation of pressure through the Cassini system, such as the following:

$$\begin{aligned} \text{open}(\text{lox}_1) \wedge \text{pressurized}(\text{lox}_1) &\implies \text{pressurized}(\text{m1\_pre\_lox}_0) \\ \text{open}(\text{lox}_1) \wedge \text{pressurized}(\text{lox}_1) &\implies \text{pressurized}(\text{m2\_pre\_lox}_0) \end{aligned}$$

So augmented, the formula would capture the non-local ramifications of opening any set of valves in any order. In practice, adding these clauses to the propositional formula constructed by Blackbox and correctly retrieving a fragment from it was a campaign waged over many weeks. For those familiar with the underlying planning technology, at issue was the planning graph that Blackbox generates. Intuitively, the planning graph is a tool for Blackbox to determine what actions are potentially relevant or definitely irrelevant at

every step in a potential plan, and what actions might possibly be used to satisfy the pre-conditions of an action that has been inserted into the plan. Construction of the planning graph assumes that the post-conditions of one action will directly satisfy the pre-conditions of another. That is, it is not equipped to determine how the ramifications of the post-conditions of one action could propagate through a model of a physical device to indirectly satisfy the pre-conditions of another action. As a result, it appears to the planning graph that there is no way to satisfy the goal condition of a planning problem. In addition, actions that are required because their ramifications set up a necessary pre-condition in the plan appear to be irrelevant because their direct post-conditions are not used by another action. To address these problems, we first create a set of temporary actions within a planning domain that mimic the effects of the background model. These actions allow the planning graph to determine that a plan to achieve the goal is possible. After the planning graph is converted to a propositional formula for solution by the SAT engine, we remove the portion of the formula that represents the effects of the temporary actions. We also insert a propositional formula that captures the non-local ramifications of the actions of the domain. This allows the SAT engine to find a plan based upon the ramification-based encoding of the planning problem. We also redesigned the plan minimization subroutine of Blackbox from a design based upon the planning graph, which would minimize actions it could not determine were relevant, to a logic-based design that used the full ramification model to determine which actions could safely be removed from a plan. We did not solve the general problem of representing ramifications within a Graphplan representation, which remains an interesting open problem.

## 7.8 A Brief Illustration

In this section, we present a small illustration of the fragment based planner operating on the Cassini propulsion system illustrated in Figure 7.11 in order to provide an intuition into its working. Detailed experimental results are provided in the next chapter. For this illustration, we consider only failures of the redundant H<sub>2</sub> valves; m1\_h2\_pre\_01 and m1\_h2\_pre\_02 on engine M1 and m2\_h2\_pre\_01 and m2\_h2\_pre\_02 on engine M2. We first consider the case where one of the redundant H<sub>2</sub> valves of M1 is permanently stuck

Time	Action
1	(lock)
2	(open=open helium_1)
3	(lock)
4	(open=open lox_1)
5	(lock)
6	(open=open m2_pre_lox_0)
7	(lock)
8	(open=open h2_1)
9	(lock)
10	(open=open m2_pre_h2_0)
11	(lock)
12	(open=open m2_pre_lox_1)
13	(lock)
14	(open=open m2_pre_h2_2)
15	(lock)
16	(open=open m2_lox)
17	(lock)
18	(open=open m2_h2)
19	(lock)
20	(open=open m2_pre_h2_1)
21	(lock)
22	(ignite=ignite m2 m2_lox m2_h2)

Figure 7.12: A Minimal Plan When M1's Valves are Stuck

closed. Any conformant plan for this example must either make use of engine M2 or open both of the possibly stuck valves of M1 . Initially, we will restrict the planner to using only the minimal number of actions needed to ignite M2 , that is 22. Thus the option of using M1 is unavailable. Only plans that open exactly the minimal number of valves needed to ignite M2 , with choices where redundancy appears, can be generated. Figure 7.12 illustrates one such plan. Note that we have removed references to the aspects of the open operator. Figure 7.13 illustrates the performance of the fragment based planner using probing on this problem. Since probing and our SAT algorithm are randomized algorithms, we show 100 trials each with a different random seed. The horizontal axis is the 100 trials in an arbitrary order. The vertical axis denotes the number of iterations required by each trial, where an iteration is one call to the SAT algorithm to generate a fragment. At first glance, this graph may not appear terribly instructive. Consider instead Figure 7.14 where we have simply sorted the random trials by how many iterations they required.

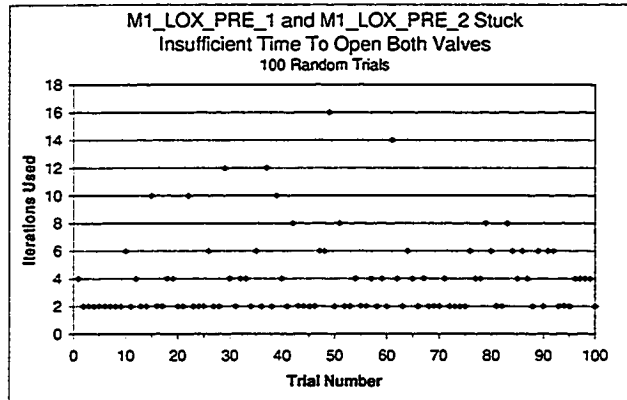


Figure 7.13: Iterations Required for Two Valves On One Engine

Figure 7.14 illustrates how the probing strategy on top of Blackbox is generating conformant plans. Note that approximately 50% of the trials require 2 iterations, then roughly 25% require 4 iterations, 12.5% require six, and so on. Intuitively, half of the trials initially choose to use M2 and succeed. The remaining half of the trials initially choose to use M1, fail and are restarted. Again, half choose to use M2, and succeed.

Let us consider this in slightly more detail. Recall that the goal of our planning problem is to produce thrust. This goal and one of the initial worlds, either `m1_lox_pre_1` being stuck or `m1_lox_pre_2` being stuck, are asserted into Blackbox. Without any loss of generality, let us assume that `m1_lox_pre_1` is chosen first. Blackbox must then generate a single world plan. Blackbox encodes the planning problem in a planning graph that plans in a regression style. Thus it captures that thrusting implies either igniting M1 or igniting M2. If M2 is chosen, a 22 step fragment that ignites M2 is generated for the world where `m1_lox_pre_1` is stuck. We assert this fragment into the planning domain, so that the plan for the next world will be forced to include it. When we consider the other world, where `m1_lox_pre_2` is stuck, we find that no additional actions are needed. If M1 is chosen for our initial fragment when `m1_lox_pre_1` is stuck, Blackbox must generate a 22 step plan that uses `m1_lox_pre_2`. This fragment is asserted. When we consider the second world where `m1_lox_pre_2` is stuck, this fragment by itself is not sufficient. We would like to augment the existing plan by opening `m1_lox_pre_1`. Unfortunately, since the planner is

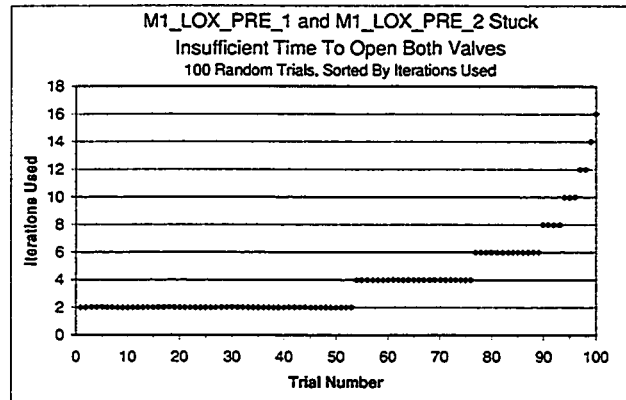


Figure 7.14: Iterations Required for Two Valves On One Engine, Sorted

constrained to use only 22 actions, no such plan is found. The previously asserted fragment using `m1_lox_pre_2` is therefore abandoned and the planning process is restarted.

Similarly, consider Figure 7.15. In this example, one valve on either engine, either `m1_lox_pre_1` or `m2_lox_pre_1` is stuck. Again, without any loss of generality, let us assume that `m1_lox_pre_1` is chosen first. Again the first fragment must choose M1 or M2. If M1 is chosen, then `m1_lox_pre_2` must be used in the initial fragment. Since `m1_lox_pre_2` is not stuck in the other world, this should account for 50% of the trials succeeding in two iterations. If M2 is chosen in the world where `m1_lox_pre_1` is stuck, then either `m2_lox_pre_1` or `m2_lox_pre_2` may be used to ignite M2. So long as `m2_lox_pre_2` is chosen, then the fragment will work in the second world, where `m2_lox_pre_1` is stuck. This accounts for an additional 25% of the trials succeeding on with two iterations, for a total expectation of 75%. In the remaining case when considering the world where `m1_lox_pre_1` is stuck, for our first fragment we choose ignite M2 using `m1_lox_pre_2`. Again, when we consider the other world we need to open an additional LOX valve on the same engine, and cannot due to the constraint on the number of actions.

Figure 7.16 illustrates planner behavior when either `m1_h2_pre_01` and `m1_h2_pre_02` is stuck closed and when more than sufficient time is allowed to open both if necessary. Out of 100 trials, 52 created a plan that used M2 for the initial fragment. When generating the second fragment, it was not necessary to add any actions. In the remaining 48 trials, M1

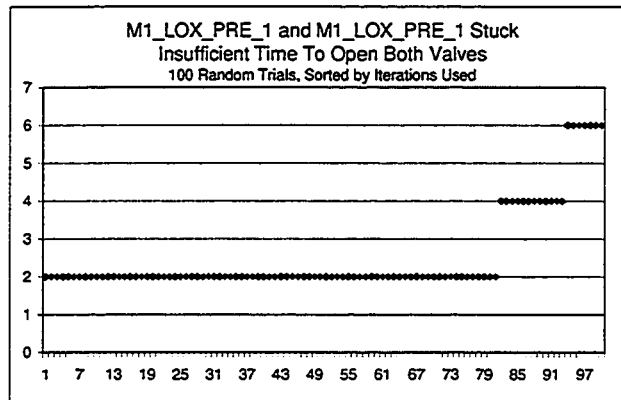


Figure 7.15: Iterations Required for One Valves On Each Engine

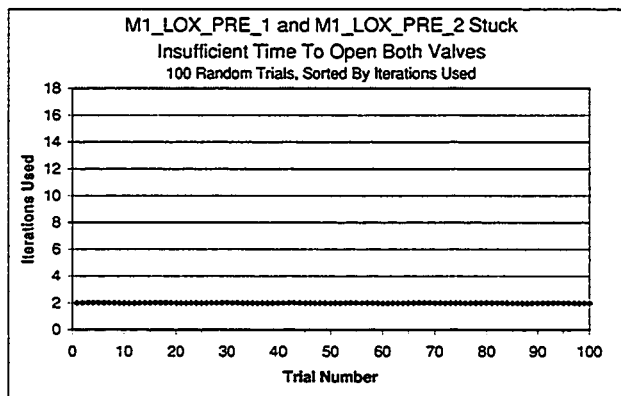


Figure 7.16: Iterations Required With Sufficient Time Steps

was chosen, and either `m1_h2_pre_01` or `m1_h2_pre_02` was opened in the first fragment. When generating the second fragment, it was discovered the first fragment was not sufficient, and an action to open the other valve was inserted, along with an unlock action. Thus, all 100 trials take 2 iterations. The next chapter includes a greater variety of experiments with the Cassini model, and a greater variety of corresponding behaviors by the fragment-based conformant planner.

## 7.9 Summary

In this chapter we have introduced fragment-based conformant planners that choose fragments that will appear in the conformant plan. Because of this choice, the planner may need to backtrack or restart. We have also introduced a conflict-based planner that finds

*noGoods* that cannot appear in any conformant plan. Because these *noGoods* are guaranteed not to appear in any conformant plan, no backtracking is needed and the simple algorithm of Figure 7.9 is complete. However, the extent to which each *noGood* actually focuses the planner and the extent to which the approach becomes a generate and test strategy is at issue. The fragment- and conflict-based approaches are compared in the results section in order to address these issues.



## Chapter 8

# Experimental Results for Conformant Planning

The planning system described has been implemented in C++, making use of existing functionality from Blackbox, Satz (Li & Anbulagan 1997), and Graphplan. In the first section below, we compare the performance of an initial version of the planner that did not include conditional actions against a number of planners from the literature using the bomb in the toilet problem and a simplification of the RING domain<sup>1</sup> (Cimatti & Roveri 2000). In the second section, we compare different search strategies for the fragment based planner on the bomb in the toilet problem and a conformant logistics problem wherein packages must be delivered via a set of roads that might contain mines. The third section investigates performance of the planner with the slight modifications needed to allow conditional actions and proper handling of ramifications, using the Cassini spacecraft as the domain. The final two sections describe the performance of a conflict-based conformant planner, and performance when we attempt to find a conformant plan by taking forming a single world that conjoins all of the possible worlds and planning with a deterministic planner.

### 8.1 Performance of the Fragment-based Planner

In the first subsection, we compare performance of *fragPlan* on the bomb in the toilet problem against planners from the literature. In the next subsection, we illustrate scalability as the number of possible worlds increases. Finally, in this section we illustrate how the performance of different search strategies varies with the domain.

---

<sup>1</sup>The full ring domain requires actions with conditional effects which, as noted above, our initial implementation did not support.

Time	Action
1	dunk package1, toilet1
2	flush toilet1
3	dunk package2, toilet1
4	flush toilet1
5	dunk package3, toilet1
6	flush toilet1
7	dunk package4, toilet1
8	flush toilet1
9	dunk package5, toilet1
10	flush toilet1
11	dunk package6, toilet1

Figure 8.1: A bomb in the toilet plan for 6 packages, 1 toilet

Time	Actions	
1	dunk package1, toilet1	dunk package2, toilet2
2	flush toilet1	flush toilet 2
3	dunk package3, toilet1	dunk package4, toilet2
4	flush toilet1	flush toilet 2
5	dunk package5, toilet1	dunk package6, toilet2
6	flush toilet1	flush toilet 2

Figure 8.2: A parallel bomb in the toilet plan for 6 packages, 2 toilets

The bomb in the toilet domain (McDermott 1987) is an oft-used benchmark for which performance information is available for many planners. In this domain, we are presented with a set of packages. One of the packages contains a bomb. In order to defuse the bomb, we must dunk it in a toilet. Once a toilet has been used to dunk a package, it clogs and must be flushed before another package can be dunked. This is referred to as the bomb in the toilet domain with clogging, or *BTC*. Figure 8.1 illustrates a successful plan for the six package, one toilet problem. Figure 8.2 illustrates a successful plan for the six package, two toilet problem. Note that when more than one toilet is available, actions may be taken in parallel. This is significant because not every planner from the literature we consider is able to generate plans that occur in parallel. Instead, they must consider a single action at a time, resulting in serialized plans such as the one illustrated in Figure 8.3.

#### 8.1.1 Performance Comparison on the BTC Domain

Figure 8.4 illustrates performance of planners that consider only serial actions versus *frag-Plan* on variations of the BTC problem. The first column lists the number of packages and

Time	Action
1	dunk package1, toilet2
2	dunk package2, toilet1
3	flush toilet1
4	dunk package3, toilet1
5	flush toilet1
6	flush toilet2
7	dunk package4, toilet1
8	dunk package5, toilet2
9	flush toilet1
10	dunk package6, toilet1

Figure 8.3: A serial bomb in the toilet plan for 6 packages, 1 toilet

<i>P-T</i>	Steps	GPT	CMBP	HSCP	<i>fragPlan</i>	
					Time	Calls
6-1	11 / 11	0.07	0.01	0.01	0.11	23.68
8-1	15 / 15	0.11	0.20	0.01	0.47	40.90
10-1	19 / 19	1.31	0.71	0.01	2.89	124.52
6-4	8 / 3	1.44	0.41	0.01	0.03	7.5
8-4	12 / 3	8.78	2.74	0.03	0.23	66.43
10-4	16 / 5	59.97	14.42	0.03	0.44	45.70
6-6	6 / 1	8.69	3.29	0.03	0.02	6.90
8-6	10 / 3	68.43	20.71	0.05	0.05	8.23
10-6	14 / 3	486.97	111.83	0.08	0.27	19.68

Figure 8.4: *fragPlan* and serial planners on BTC

toilets in the problem. The second column lists the minimum number of time steps in a conformant plan for planners that consider a single action at a time, and the minimum number of time steps required for planners that allow parallel actions. The next three columns of the table show results for three planners from the literature. GPT (Bonet & Geffner 2001) is a planner that uses A\* search and dynamic programming to solve conformant, contingent and probabilistic planning problems. CMBP (Cimatti & Roveri 2000) is a conformant planner that uses binary decision diagrams (BDD) to represent the outcome of candidate actions in all worlds simultaneously. HSCP (Bertoli, Cimatti, & Roveri 2001) is an impressive successor to CMBP that uses a heuristic to control the actions considered during search. Figures for these planners were taken from (Bertoli, Cimatti, & Roveri 2001) and were generated on an 300MHz Pentium II PC running Linux with 512M of memory. The final two columns illustrate the performance of *fragPlan* using stochastic probing as the search strategy. The first column lists the time to solve the problem. Since we are using a randomized procedure, timing figures are averages over thirty runs. The next column shows the average number of calls to the Plan procedure to find a fragment. *fragPlan* was run on a 266MHz Pentium II laptop running Windows 2000 on 288M of memory. Figure 8.6 illustrates performance of C-plan (Castellini, Giunchiglia, & Tacchella 2001), a conformant planner that like *fragPlan* uses a propositional representation that allows parallel actions. C-plan generates possible plans that may be conformant and tests whether each is in fact conformant. The label TIME indicates C-plan did not find a plan in 1200 seconds. The second column under C-plan lists the number of possible plans that are tested for each problem. Figures for C-plan were taken from (Castellini, Giunchiglia, & Tacchella 2001) and were generated on an 850MHz Pentium III PC running Linux with 512M of memory. Relative to these experimental runs, we have made several observations.

**Observation 1** *On serial BTC problems, fragPlan is competitive with GPT and CMBP but is dominated by HSCP.*

Consider the first three rows of Figure 8.4 which represent problems with multiple packages and one toilet, {6-1, 8-1, 10-1}. These are completely serial problems, in that there is a single toilet, and only one package can be dunked at a time. The task for *fragPlan* is

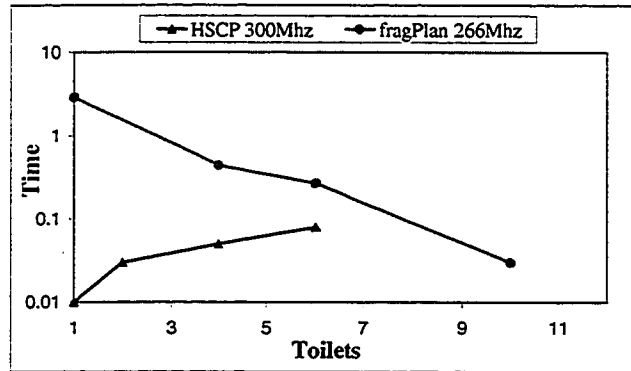


Figure 8.5: Time versus parallelism on BTC for fragPlan

to order the dunking fragments such that the final plan alternates dunking and flushing within the minimal number of time steps. In the six package problem, probing generates an average of 23.68 calls to Plan rather than the minimal 6. This indicates there is some amount of misplacement of fragments which is being addressed by restarting the algorithm. As we increase the number of steps by increasing the number of packages, the amount of restarting required increases. This is intuitive, as there are more opportunities for fragments to be misplaced. However, the number of fragments generated and the difficulty of finding those fragments, as judged by the total runtime of the algorithm, appear to combine to scale at approximately the same rate as problem complexity in the GPT and CMBP planners. HSCP uses the same BDD representation as CMBP, but uses a heuristic function rather than blind search to select actions. This heuristic appears to do an excellent job of focusing the search, and HSCP is significantly faster than *fragPlan* on these serial problems.

**Observation 2** *On parallel BTC problems, fragPlan dominates CMBP and GPT, and eventually surpasses HSCP.*

Like CMBP, HSCP cannot generate parallel plans in domains where they are required, but will produce serialized plans. Consider for example the performance of CMBP on the problems with ten packages. CMBP encodes the set of initial states of the world in a BDD, then considers the outcome of the possible actions on each world in the set. The result of the action on each possible world is a set of worlds that is also encoded in a BDD. When each world in a BDD satisfies the goal, a plan has been found. As the number

$P - T$	Steps	<i>fragPlan</i>		CPlan	
		Time	Calls	Time	Plans
6-1	11	0.11	23.68	221.55	52561
8-1	15	0.47	40.90	TIME	-
10-1	19	2.89	124.52	TIME	-
6-10	1	0.02	6	0.01	1
8-10	1	0.02	8	0.01	1
10-10	1	0.03	12.19	0.04	1

Figure 8.6: *fragPlan* and C-plan on BTC

of toilets increases, so does the number of actions that must be applied to each BDD. The search branching factor in the 6 toilet problem is 6 times larger than in the 1 toilet problem, while the search depth is less effected since CMBP does not consider parallel actions. This leads to an explosion in the number of BDD's that CMBP must generate. Thus execution time increases from 0.71 seconds to 111.83 seconds. In the case of GPT, we are not certain of the exact mechanism behind the increase in time. We suspect it is a similar explosion in the number of possible partial plans and resulting states of the world it must consider as it performs an A\* search over action sequences. The search heuristic in HSCP again does an excellent job of focusing the search, so execution time increases by only an order of magnitude between 1 and 6 toilets. In contrast, the execution time for *fragPlan* *decreases* by two orders of magnitude between 1 and 10 toilets. Figure 8.5 illustrates the drop in runtime for *fragPlan* on the 10 package problem as the number of toilets increases, and the corresponding growth in HSCP. Intuitively, *fragPlan* is attempting to assemble the fragments necessary to achieve the goal in each world along with the necessary inter-fragment repair actions, within the allotted number of time steps. Allowing parallel actions significantly simplifies the problem of aligning the necessary fragments to the appropriate time steps.

**Observation 3** *On BTC, fragPlan dominates C-plan.*

Like *fragPlan*, C-plan generates parallel plans. Turning to Figure 8.6, C-plan appears to be at a severe performance disadvantage. Conceptually, in order to find a plan of length  $n$ , C-plan tests every possible plan of length less than  $n$  as well as every possible plan of length  $n$ . C-plan adopts several strategies to reduce the number of possible plans it considers but the

number remains considerable. It is not able to find a plan in a competitive amount of time for a serial problem or moderately parallel problems, where the length  $n$  and the number of plans at each length are relatively high. See (Castellini, Giunchiglia, & Tacchella 2001) for additional performance figures. However, when the parallelism is sufficient to reduce the number of planning steps to one, *C-plan* does exceedingly well. The first and only possible plan it generates dunks all packages simultaneously, which is a valid conformant plan.

### 8.1.2 Performance with an exponential set of worlds

A common criticism of planners that explicitly enumerate worlds is that there may be an exponential number of such worlds. In the BTC domain, a problem of  $n$  packages has  $2^n$  worlds if we allow that every package may contain a bomb. In the modified RING domain, MRING, a maze contains  $n$  rooms. Each room has a window that may be open, closed or locked, yielding  $3^n$  worlds. The goal of MRING is for all windows to be locked, and a robot placed at a known location may close or lock the window in the current room, or move to the next room. Thus the maximum number of worlds is dictated by the number of bombs in BTC and the number of unlocked or open windows in MRING. We have performed a number of experiments that vary the number of worlds in the BTC and MRING domains and make the following observations.

#### **Observation 4** *The representation size is constant*

Unlike planners such as CGP, fragPlan does not duplicate its planning representation for each world. The propositions capturing each world (*e.g.*, whether or not package  $i$  has a bomb) are asserted into the representation in turn. Thus the memory required by fragPlan is dominated by its single world Blackbox representation and does not increase during search. For problems where the worlds are combinations of a fixed number of properties, the only increase in memory to consider additional worlds is a few bytes per world to represent the corresponding fragment.

#### **Observation 5** *Plan calls approach the number of worlds*

As the number of possible worlds increases for many domains, there tends to be a great deal of overlap between worlds. Often the large number of worlds is the result of the

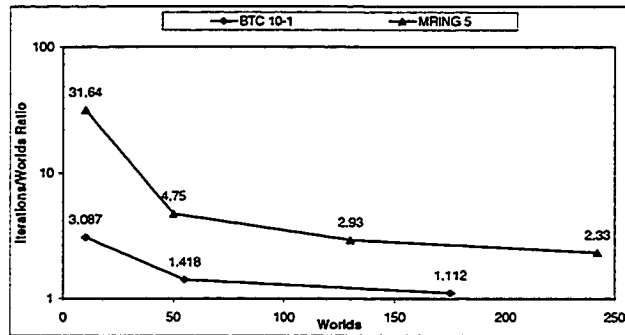


Figure 8.7: Effect of Worlds on Iterations Per World for fragPlan

combination of a smaller set of underlying conditions. In this case, conformant plans for some of the worlds will often be conformant plans for many other worlds. For example, any fragment that solves the BTC world where an entire set of packages have bombs, is a conformant plan for worlds where any subset of those packages have bombs. Similarly for the Ring domain - a fragment that solves the world where a set of rooms have open windows will be a conformant plan for worlds where any subset of those rooms have open windows. Note that this characteristic is not limited to artificial domains such as BTC and Ring. Consider a problem where there are  $N$  candidate faults in a spacecraft, and one must plan to achieve a goal despite these possible faults. Plans for worlds in which several of the faults are present typically work for worlds in which a subset of those faults are present. As a result, even though the number of possible worlds is growing exponentially with the number of independent sources of uncertainty, the planner tends to discover a conformant plan after considering only a few of these worlds. This observation is confirmed by Figure 8.7, which shows the ratio of plan calls required for the BTC problem with 10 packages and MRING problem with 5 rooms as the number of possible worlds increases. For all domains that we have tested, this ratio approaches 1 as the amount of uncertainty in the domain is increased.

In other words, for most of the worlds, the planner is just verifying that the current plan works on the current world. We note that this verification problem is polynomial, whereas the planning problem for a single world given a fixed horizon is NP-complete. As a result, there is no a priori reason to expect that verification on an exponential number



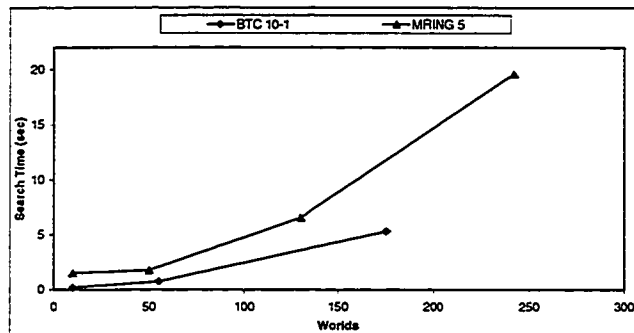


Figure 8.8: Effect of Worlds on fragPlan Search Time

of worlds is computationally worse than planning on a single world. However, our current implementation does not take advantage of the polynomial nature of verification. As shown in Figure 8.8, we do see growth in search time as the number of worlds increases, but this growth is very slow.

Of course, there is no guarantee that for problems with a large number of worlds the fragment generated for one of the worlds will satisfy any of the other worlds without additional actions. First, a domain may present a very large number of independent worlds that are not combinations of a set of underlying conditions. The actions required for each world might therefore be unique. Even in cases where worlds are generated in a combinatoric fashion, our intuitions about how the plans for related sets of worlds are related to each other may not hold. Consider the example where worlds consist of members of the powerset of a set of failures. It's relatively plain that the concatenation of the conformant plans for each of two failures  $f1$  and  $f2$  may not be a valid plan when both failures are present. For example, if  $f1$  and  $f2$  represent the failure of two redundant components that provide the same service, then the plan for the failure of either is to enable the other. The composition of these two plans, that is to enable both, is not a plan if both are failed. We may still get significant traction by generating a plan for large combinations of failures that also applies to any subset. Note however that this is not necessarily the case. The presence of some failure  $f$  may make planning for any set that includes  $f$  significantly easier than the same set without the existence of  $f$ . Intuitively, consider the case where we cannot diagnose whether an electronic component has hung and needs to be turned off and on, or

its power switch has been knocked into the off position. The plan for the combination of both failures is to simply turn the power on, which is not a plan for either failure considered separately.

## 8.2 Comparison of Search Strategies for fragPlan

Of the several search strategies we implemented for fragPlan, our experiments yielded the most interesting results for stochastic probing and bubbling. Stochastic probing in essence selects a world ordering, chooses a fragment for each world, and restarts as soon as a suitable fragment cannot be found. Bubbling attempts to solve the most difficult worlds first, then add back in worlds that appear increasingly difficult. In general in all of our experiments, it was difficult to beat stochastic probing by a significant margin. In cases where the initial worlds displayed significant differences in difficulty, bubbling was able to best probing by a small multiple in performance. The details of these experiments and further performance insights are given below.

### **Observation 6** *Stochastic Probing Dominates on BTC*

A fragment set may fail to extend into a conformant plan because of a global property of the partial plan rather than a property of any particular fragment. In these cases, we expect bubbling's attempt to solve the most difficult worlds first based upon plan failures will not to lead to a conformant plan. As illustrated in Figure 8.22, the BTC problem has this property. When the fragment for the final world  $w_j$  cannot be placed, it is because of the placement of all of the existing fragments. Bubbling removes the fragment for the previous world considered,  $w_i$ . This allows the fragment for  $w_j$  to replace the fragment for  $w_i$ . The roles then reverse, making bubbling completely ineffective on the BTC problem. Outfitting bubbling with a random restart or asserting previously attempted fragment sets as nogoods would prevent this type of cycle from developing. More generally, locally re-ordering the worlds does not guarantee a plan.

### **Observation 7** *Bubbling Dominates on Asymmetric Worlds*

In order to further compare search strategies, we defined a logistics problem with uncertainty in its initial conditions. We used this problem to investigate the advantage of probing

Worlds	Relevant Mines	Irrelevant Worlds	Average Calls to Plan	
			Probing	Bubbling
6	8 in $w_1$	5	33.52	11.39
6	4 each in $w_1, w_2$	4	44.81	12.61
70	1 each in $w_1 - w_5$	65	510.83	128.93

Figure 8.9: Probing vs. Bubbling on Asymmetric Worlds

on problems where a small subset of the worlds are more difficult to plan for than the rest.

**Example 10** Consider the problem of delivering relief packages to refugee camps. A depot with packages is located in one location and a number of camps are at distinct locations. Two locations may be connected by an incoming and an outgoing route. One delivery truck and one minesweeper are available. A subset of the routes may be mined.

A plan for this problem must run the minesweeper between the truck's initial location and the depot, drive the truck to the depot, load the truck, and so on. Figure 8.9 illustrates performance on three cases of this problem. In each problem, five camps require packages and one does not. The first column specifies the number of worlds, where a world is an assignment of mines to routes. In order to create structure in the worlds, not all worlds specify that routes needed to achieve the goal are mined. In the second column world  $w_1$  represents the belief that 8 relevant routes are mined. In the remaining worlds, specified in the third column, the mines are on routes irrelevant to the goal. The final row is an exaggerated case wherein 65 worlds specify mines on a route that is not needed in the plan, and camps are in a linear arrangement that maximizes the number of mines that must be cleared in order to reach the most difficult (farthest) camp. This problem is particularly suited for bubbling, as we must clear the mine on the first route before considering driving to the second route and clearing it. The fourth column denotes the average number of calls to Plan that are made before finding a conformant plan, averaged over 30 plans. While probing does not perform as well as bubbling, even on the extreme case wherein 5 of 70 worlds must be considered in order, it does not do as poorly as we expected. The next observation provides further insight into this behavior.

**Observation 8** *Plans for one world are often generated so as to work in others*

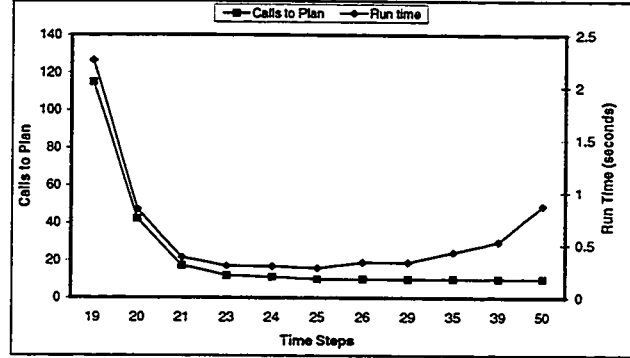


Figure 8.10: Effect of Time Steps on fragPlan

Given the number of worlds in Figure 8.9, we were surprised that probing did not perform significantly worse. We discovered that plans for world  $w_i$  returned by Plan often work in some other world  $w_j$ . We think of this as *lazy conformance*. Our SAT encoding specifies the impact of every world upon reaching the goal, even though only one world is asserted when planning. We believe this biases the heuristics of the SAT procedure to activate actions that remove threats to the goal under consideration from  $w_j$  when finding a plan for  $w_i$ . Intuitively, the heuristics of the SAT procedure see a threat to a precondition of the goal without seeing that the threat is only entailed if  $w_j$  is asserted. Even if *extractFragment* removes the extra actions for world  $w_j$  from the fragment inserted into the planning domain, they ensure that there is space for such actions to be easily added when  $w_j$  is considered. The result is that in this domain fragment-based planning is less sensitive to irrelevant worlds, fragment placement and world ordering than we would have expected.

**Observation 9** *Probing improves with longer horizons*

When we use a randomized search within fragPlan, we expect performance to be more sensitive to how tightly the fragments constrain each other than by the total number of action sequences. Figure 8.10 illustrates the performance on BTC with 10 packages and 1 toilet as the planning horizon expands from 19 steps to 50. Note the calls to Plan decrease to the minimum of one per world while the number of action sequences increases exponentially. Intuitively, placing 10 dunk fragments becomes significantly easier with a few steps of slack. Eventually, the penalty for manipulating a larger representation outweighs the

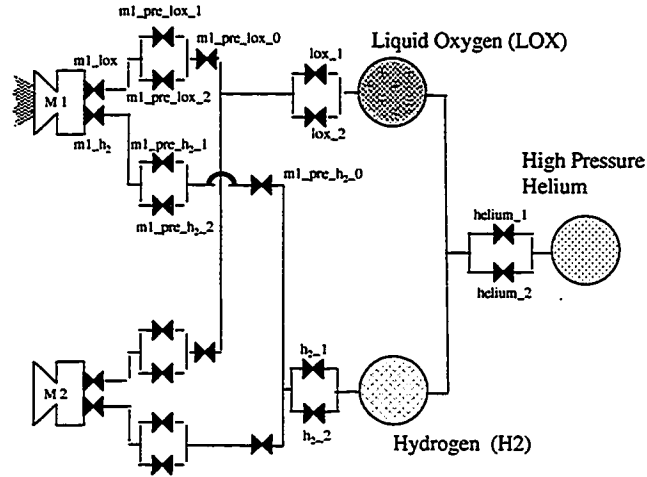


Figure 8.11: Cassini Propulsion System Schematic

reduction in search. However, even at 50 steps performance is good. We have documented this phenomenon on all domains we considered. For control applications where some plan must be found, this suggests quickly generating a plan given a conservative planning horizon then iteratively shortening the horizon to find shorter plans in the time that remains.

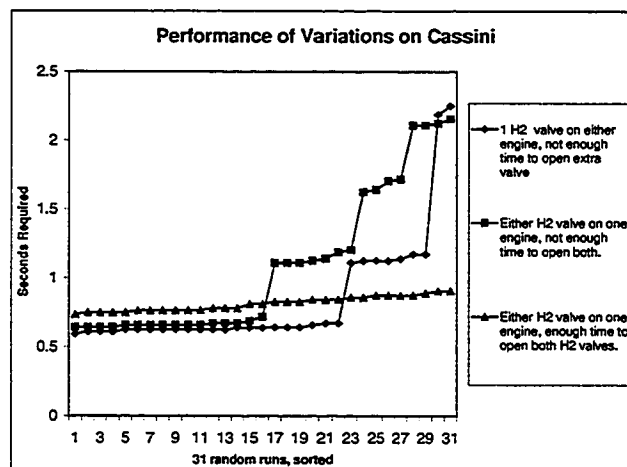


Figure 8.12: Performance of 31 trials of fragPlan on 3 Cassini problems

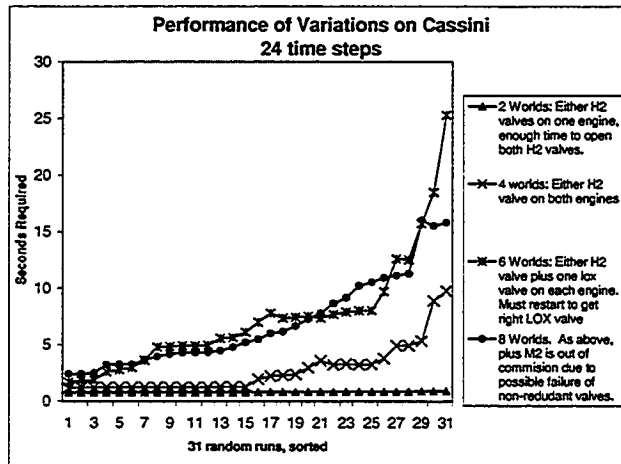


Figure 8.13: Performance of 31 trials of fragPlan on 4 Cassini problems

### 8.3 Performance on the Cassini Domain

Figure 8.11 illustrates the Cassini propulsion system described in preceding chapters. We have performed experiments with fragPlan on a wide variety of problems involving this domain. In each problem, the goal is to produce thrust. This can be achieved by igniting either main engine M1 or M2 after opening the appropriate valves to feed in  $H_2$  and LOX. Each conformant planning problem has multiple worlds, each consisting of a single valve that may be stuck. In some problems the stuck valves only effect one engine, so the simplest conformant plan to produce thrust simply uses the other engine. In some cases, thrust can be achieved by opening both of a pair of redundant valves on one engine. In some cases, a non-redundant valve may be stuck on either engine, requiring a conformant plan that attempts to ignite both engines in order for one of them to ignite and produce thrust. All of these experiments were performed on a 1.7GHz Pentium 4 running Windows XP with 512MB of memory, though only a fraction of this memory was used by fragPlan.

Figure 8.12 recapitulates the three scenarios discussed at the end of Chapter 7. Each of these scenarios has two initial worlds. In the first scenario, either engine may have one redundant  $H_2$  valve stuck. Since there are not enough time steps in the planning horizon to open a stuck valve and its redundant counterpart, the only conformant plan is to use an  $H_2$  valve that is not stuck in any world. As discussed in Chapter 7, approximately 75% of the

trials choose such a plan as the first fragment. The rest are solved after the generation of a few additional fragments. In the second scenario, the planning horizon is similarly limited, but both possibly failed valves are on a single engine. Thus, as discussed in Chapter 7, 50% of the trials choose a conformant plan for their first fragment. In the third scenario, both possibly failed valves are again on a single engine but there is sufficient time to open both of the redundant  $H_2$  on the engine. Thus the fragment chosen for the first world can easily be extended to handle the second world. Conformant plans for this scenario are to open both  $H_2$  valves above the engine that may have a failed valve, or to ignite the other engine.

Figure 8.13 illustrates performance for an increasing number of worlds, with the planning horizon limited to 24 time steps. The two world scenario is equivalent to the third scenario above. Either redundant  $H_2$  valve on a single engine may be stuck. In the four world scenario, any of the four redundant  $H_2$  valves above the two engines may have failed. Here we must open both redundant valves above one of the engines before igniting it. In the six world case, any of the redundant  $H_2$  valves may be stuck, plus one of the redundant LOX valves on either engine may be stuck. There is insufficient time in the planning horizon to both open two redundant  $H_2$  valves and two redundant LOX valves. Thus a conformant plan must open both redundant  $H_2$  valves for an engine, and open the redundant LOX valve that is not failed in any world. If a fragment opens a LOX valve that may be failed, fragPlan must restart, leading to the long run times of a fraction of the trials for the six world scenario. Finally, in the eight world scenario, two of the non-redundant valves of M2 may be stuck. Thus any fragment that uses M2 will cause fragPlan to restart once a world containing such a failure is encountered.

The most interesting aspect of running fragPlan on the Cassini domain is its behavior as we vary the number of time steps in the planning horizon. The results were quite unexpected. Recall that with the bomb in the toilet problem, the problem gets easier as we add time steps. As shown in Figure 8.10, the running time and number of planning iterations required for the BTC problem drop precipitously as we add time steps to the planning horizon. This is because the problem of fitting the fragments into the planning horizon is becoming less constrained. Figure 8.14 illustrates the relationship between the length of the

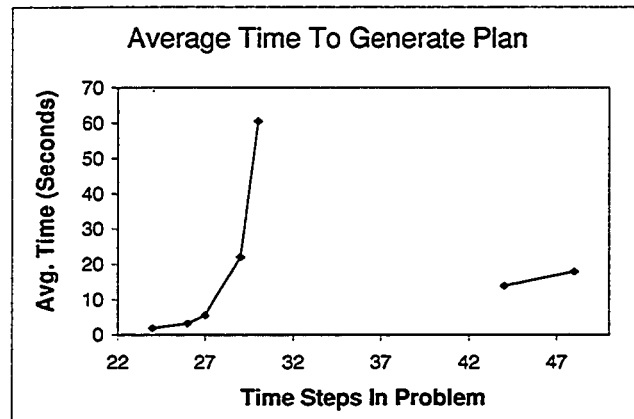


Figure 8.14: Effect of Time Steps on fragPlan time for the Cassini domain

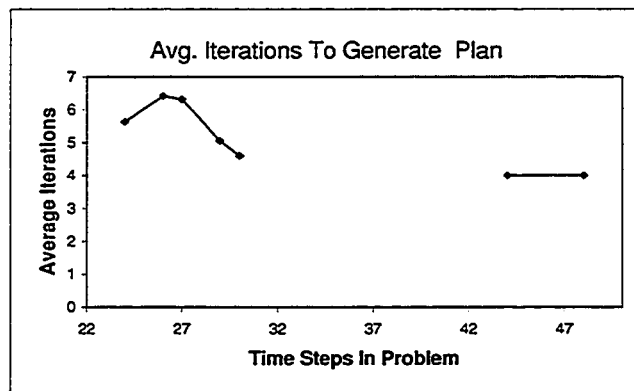


Figure 8.15: Effect of Time Steps on fragPlan iterations for the Cassini domain

planning horizon and average run time for fragPlan, taken over 100 trials, on the Cassini domain. This is for a problem where for each engine, either of the redundant  $H_2$  valves may be stuck, yielding four worlds. At the minimal number of time steps, 24, fragPlan is relatively fast, taking approximately two seconds per trial. As we increase the planning horizon, the average time to generate a conformant plan rises precipitously. Between 30 and 42 time steps, inclusive, performance becomes so poor that we did not seek to run 100 trials. Trials would intermittently exceed 300 seconds each before being terminated without a plan. Curiously, at 43 time steps, the average time drops precipitously to under 10 seconds. This bore further investigation.



There are three scenarios under which planning time might explode; the number of fragments being generated in order to find a plan explodes, the average time required to generate a fragment explodes, or both. Figure 8.15 illustrates the same sets of 100 fragPlan trials per planning horizon as Figure 8.14, but displays the average number of iterations (that is, fragments generated) over 100 trials rather than the average running time. Note that as we increase the planning horizon, the number of iterations is decreasing as the problem of fitting fragments into the horizon becomes less constrained. By 43 time steps, all 100 runs take the minimal number of iterations possible, one per world. By comparing Figure 8.15 and Figure 8.14, we can see that as the horizon lengthens from 24 to 30, the average time to generate a fragment must be exploding, while above 43 time steps it drops rapidly. This phenomenon is easily explained by the differences in the Cassini and BTC domains, and the way our single world planner for generating fragments, Blackbox, operates.

The planning graph that Blackbox creates forces Blackbox to do regression style planning. That is, it starts from the goal, finds an action that achieves the goal, then makes the preconditions to the action the new goal. Our goal is to produce thrust. This can be achieved by igniting either M1 or M2. In Blackbox, the planning graph is turned into a satisfiability problem, and this choice is actually performed by a SAT engine. In our case, this is Satz. Thus Satz will choose the action that ignites M1 or M2. The planning graph encoding will then force it to satisfy the appropriate preconditions, working its way through a set of choices to provide  $H_2$  and LOX to the chosen engine. Intuitively, the length of the planning horizon has a significant impact on how difficult it is for Satz to determine that it has made the wrong choice.

Without any loss of generality, consider the case where Satz chooses to ignite M1 for the first fragment. It must then open a set of valves that supply  $H_2$  and LOX to M1. These actions are then inserted into the planning domain so they will occur in all subsequent fragments. Now, fragPlan must plan for the second world. Again, the goal is to produce thrust. Note that although igniting M1 is asserted into the planning domain, the actions asserted into the domain are not sufficient to produce thrust in the second world. Thus, Satz must again satisfy the thrusting goal. Again it may choose to ignite M1, and thus in

some sense repair the plan to use M1, or it may choose to add the action that ignites M2. Let us focus on the case where it chooses to ignite M2 for the next fragment.

If there are more than 43 time steps, there are a sufficient number of actions available to open all of the valves necessary to ignite M2 in addition to the actions that the first fragment inserted to ignite M1. Thus the SAT problem is quite simple, and the time per SAT call is low. If there are less than 43 time steps, then it is impossible to ignite M2. Igniting M1 takes 22 time steps, which leaves insufficient actions to ignite the second engine. However, the number of actions available determines how much exploration Satz can do attempting to ignite M2, which is the critical problem. At 24 time steps, there are two actions available and simply asserting the action to ignite M2 effectively uses up all remaining actions. Thus there are no actions Satz can consider to meet the preconditions for igniting M2. It quickly turns to the alternative of achieving thrust by igniting M1, which requires augmenting the actions that achieve its preconditions to be applicable to the current world. As the horizon grows, so grows the number of time steps that the fragment for igniting M1 leaves unused. Satz cannot reason that the number of available actions is insufficient to fit a plan to ignite M2. It can therefore consider an explosive number of potential plans for meeting the preconditions of igniting M2 as the number of actions available increases.

Note that this problem does not occur in the BTC problem. In that problem, the plan to achieve the goal in any given world is trivial. Each world specifies which package has the bomb, and that package must be dunked. Thus it's trivial to determine if a plan for a world can be fit within a set of fragments asserted for previously considered worlds. In the RING domain, only a small number of actions are applicable at each time step. The robot may close or lock the window in the current room, or move to the next room. The planning horizons are also relatively short, curtailing the amount of fruitless searching Satz can do before it retracts a previous choice or fails. In the Cassini example, there are 14 valves that can be opened at each time step, and when no plan is available Satz may be searching in vain among candidate plans of length more than 20.

A practical solution to this problem is to limit the amount of time each Satz iteration can consume within a fragPlan trial. Figure 8.16 illustrates the performance of fragPlan when

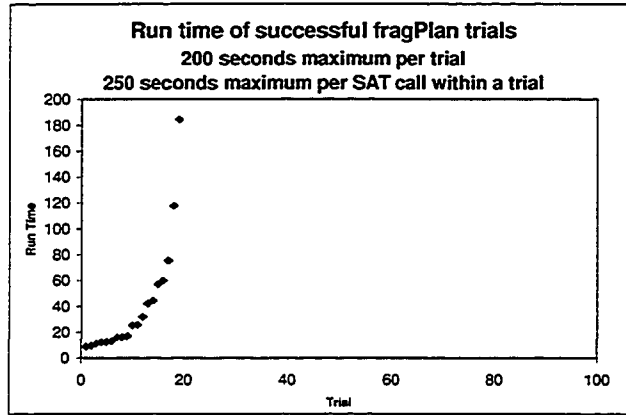


Figure 8.16: Runtime of fragPlan on Cassini with total time<200 and SAT limit<250

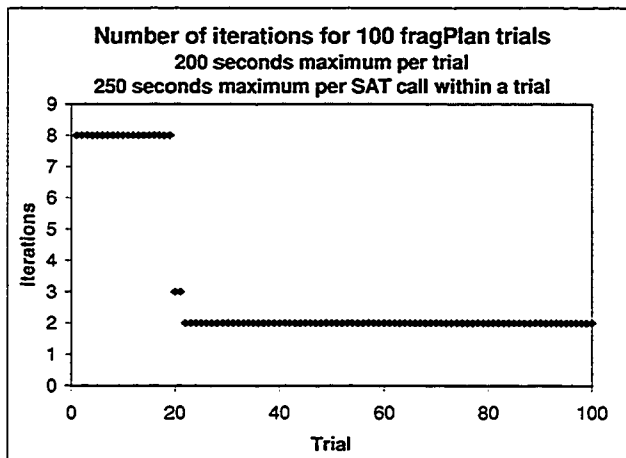


Figure 8.17: Iterations of fragPlan on Cassini with total time<200 and SAT limit<250

there is effectively no limit on the amount of time Satz may consume. Each fragPlan trial is limited to 200 seconds, and each Satz iteration is limited to 250 seconds. Thus a single bad Satz run, as described above, may consume all time available for planning. The graph illustrates the run time of 100 trials of fragPlan on an eight world Cassini problem, sorted by run time. In this domain, any of the redundant valves of the two engines, H<sub>2</sub> or LOX, may be failed. The time horizon is 35 steps, in the middle of the region where we could not effectively find plans for our simpler six world problem. Only 19 of the 100 trials are able to return a plan within 200 seconds. Times for the remaining trials exceeded 200 seconds. Figure 8.17 shows the trials in the same order, but illustrates the number of iterations in each of the 100 trials. Note that in the majority of the failed trials, only two attempts are made to find a fragment. The first finds a plan to ignite one of the engines. As described above, the attempt to generate the second fragment expends all remaining time in Satz, exploring candidate plans that fail.

Figure 8.18 illustrates the increase in performance when each Satz invocation is limited to 20 seconds. Now 66% of the trials return a conformant plan. Intuitively, when Satz chooses a subgoal that cannot be achieved and begins exploring alternative candidate methods for achieving it, only a small portion of the time remaining for planning is consumed before Satz gives up. Figure 8.19 shows the number of iterations performed by each trial. Note that in the failing trials, approximately 20 iterations are performed, even though each iteration has a maximum time of 20 seconds and the maximum total time available is only 200 seconds. Each attempt to find a conformant plan may successfully generate several fragments in significantly less than 20 seconds before Satz fails, bringing the average time per iteration to significantly less than 20 seconds.

Figure 8.20 illustrates the performance of fragPlan on this problem as the time limit per Satz call is varied. Each point in the graph represents the percentage of trials out of 100 that successfully returned a plan in 200 seconds. Note that the horizontal axis, the limit on Satz, is logarithmic. At 0.5 seconds per iteration, Satz simply cannot perform enough search to find a valid consistent of fragments in any of the trials. From 2 seconds to 20 seconds maximum per Satz attempt, performance is very stable, varying between 62% and

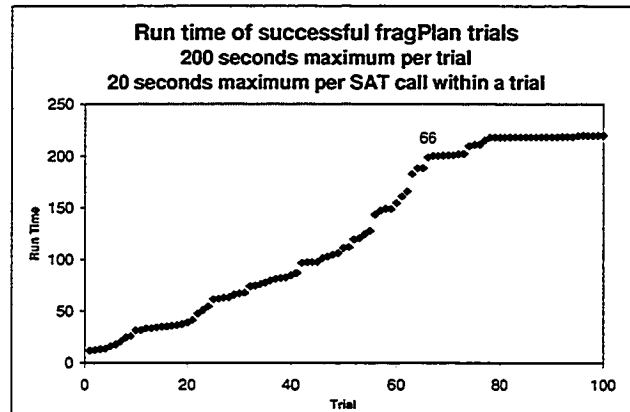


Figure 8.18: Runtime of fragPlan on Cassini with total time<200 and SAT limit<20

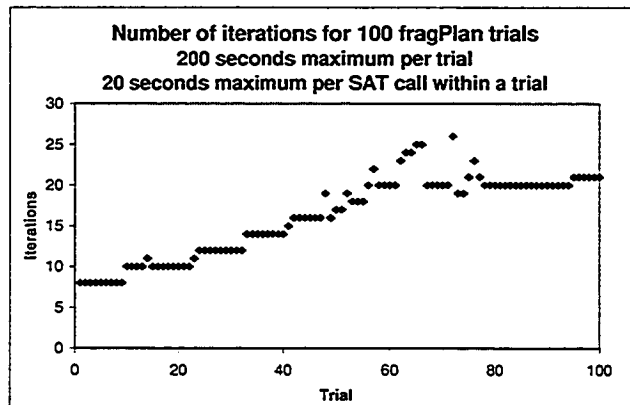


Figure 8.19: Iterations of fragPlan on Cassini with total time<200 and SAT limit<20

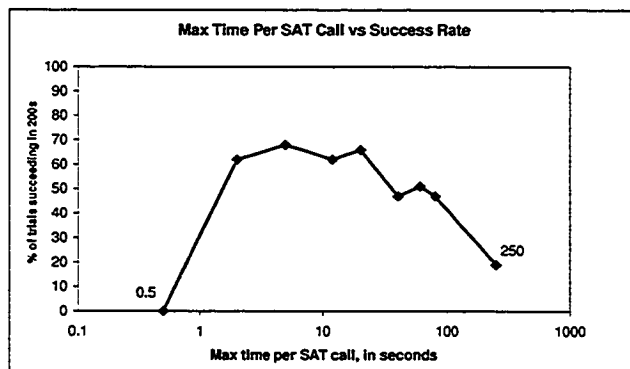


Figure 8.20: Effect of Time Steps on fragPlan time for the Cassini domain

68% success. It's interesting to note that in the 2 second case, successful trial perform between 8 and 105 iterations in 200 seconds, while in the 20 second case, successful trials perform between 8 and 25 iterations in 200 seconds. The performance is nearly identical (a 62% success rate versus 66%). We are in essence trading off a larger number of shallow Satz searches against a smaller number of more extensive Satz searches, with little change in the rate of success. As we increase the amount of time per Satz run, the penalty for a Satz run failing increases, and performance drops.

#### 8.4 Performance of Conflict Planner

The conflict based conformant planner does not appear to be useful on the BTC problem or the logistics problem. In retrospect, this is not surprising. The conflict-based planner plans in one world,  $w_1$ , then checks whether the plan achieves the goal in each other world,  $w_i$ . If the plan does not achieve the goal, the planner finds a conflict, the actions in the plan that imply the negation of the goal in  $w_i$ . Obviously, no conformant plan may contain the conflict, which is inserted into the domain to focus the planner on conformant plans. We expect this technique to be successful in focusing the planner when each conflict is relatively small and rules out a large number of non-conformant plans. We discovered two simple reasons why this fails to happen in both the BTC and logistics domains.

First, it is often not the case that a small set of actions included to achieve the goal in one world prevent the goal from being achieved in other worlds. More often, it is the lack of actions that prevent a plan from being conformant. Consider the plan illustrated in Figure 8.21. It dunks package1 at the first time step and takes no further actions. Suppose we test this plan in the world in which the bomb is in package2. There is no specific action or small set of actions that conflicts with our goal of the bomb being disarmed. The bomb is armed at the end of the plan because it was armed at the beginning of the plan and each action in the plan fails to disarm it. Thus our analysis of why the plan failed in this world returns the entire plan as a conflict and asserts its negation into the theory. Rather than ruling out a space of plans that contain some small disallowed fragment, this has the effect of inserting an enormous conflict into the theory that rules out only this single plan. Conflict-based conformant planning thus comes to resemble unguided generate and test

Time	Action
1	dunk package1
2	no action
3	no action
4	no action
5	no action
6	no action
7	no action
8	no action
9	no action
10	no action
11	no action

Figure 8.21: A Plan For A Single World

method. The results are similar in the logistics domain. If a plan sends a truck over a route that has not been cleared, the plan will fail in the first world where the route is mined. However, the action of driving on the route is not a conflict. The entire plan prefix up to driving on the route is a conflict, as replacing any action in the prefix with an action to clear the route would result in the truck safely crossing the route. The conflict rules out one specific way of blowing up the truck by not clearing the route, while leaving the planner to explore all others. Second, attempts to find conformant plans by incrementally planning one world at a time often do not fail because of conflicts between actions taken in each world. Instead, they fail because the time steps remaining unassigned in the partial plan are not distributed in the correct way to fit the remaining actions. Consider Figure 8.22. We need to fit the dunk and flush actions for package 6 into the plan, but the remaining unassigned spaces for actions do not allow it. Intuitively, the plan fails because the action timings are incorrect, rather than because of a conflict between any action and achieving the goal in any world. Thus we cannot extract any useful conflict information that would help to rule out a large space of non-conformant plans.

Figure 8.23 explores the effect of conflicts on the relatively trivial problem of finding a conformant plan for the 6 package, one toilet problem when allowed 14 time steps rather than the minimal 11. In Figure 8.23 each point on the graph represents one conformant plan that was successfully found out of 30 attempts limited to 15 seconds each. The vertical axis denotes how much time was required to find each plan. The horizontal axis numbers the

Time	Action
1	dunk package5
2	flush
3	dunk package4
4	flush
5	dunk package3
6	flush
7	
8	dunk package2
9	flush
10	
11	dunk package1

Figure 8.22: A Simple Fragment-based Plan That Failed

plans. The line for 120 conflicts indicates performance of the conflict-based algorithm with a cutoff of 120 conflicts. A plan is generated in one world and tested in the others. If it is not conformant, a conflict is generated (again, a negation of the plan that specifies that this exact plan should not be created) and added to the propositional planning model. Only the 120 most recent conflicts are kept in the theory. The line for no conflicts indicates performance of the conflict-based conformant planner run with a limit of zero conflicts. This results in pure random sampling. A plan is generated for a world chosen at random. It is then tested for conformancy. If it is conformant, it is returned. If not, no conflict is created and a new plan is generated. The line for simple fragment-based is a simple fragment-based planner with no backtracking, as shown in Figure 7.5. This planner chooses an ordering for the worlds and generates fragments in that order. This takes a fixed amount of time, but may not return a conformant plan. However, the problem of finding an 11 step plan in 14 time steps is sufficiently under-constrained that this approach never fails.

Note that allowing zero conflicts is in essence taking sampling into the space of valid single world plans. Each of these plans contains between 1 and 7 dunkings, separated by flushes. The conflict-based sampling with no conflicts performs generates approximately twice as many candidate plans as the conflict-based search with 120 conflicts, as it is not complicating the planning model with long conflict clauses. The faster, unbiased sampling with zero conflicts finds a plan 50% of the time to the 120 conflict planner's 83%. The non-backtracking fragment planner finds a plan in all 100% of the trials. These results



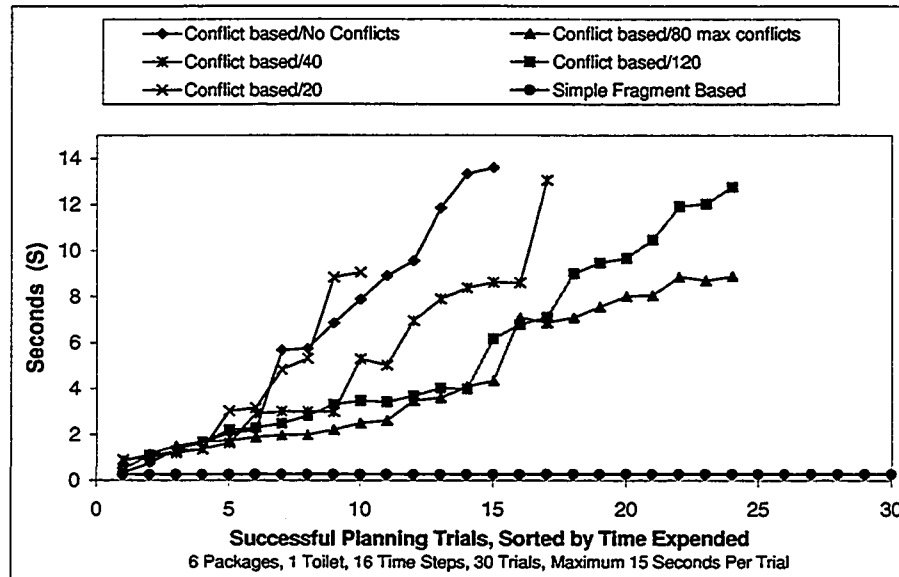


Figure 8.23: Bomb in the Toilet Performance, Limited to 15 Seconds Per Attempt

are not surprising given the non-informative nature of the conflicts in the bomb in the toilet problem. A more interesting question is why the conflicts provide even the 60% improvement over random sampling that they do. We have verified that the conflicts are simply the negation of each candidate plan, and thus should simply keep the stochastic planner from generating the same plan twice. We performed a large number of experiments to investigate whether the addition of the conflict clauses was influencing the heuristics in the stochastic SAT solver, but have not yet gained any strong insights. Given the poor performance of the conflict-based planners, we have not placed a priority on this line of work.

## 8.5 Performance of A Conjunctive Conformant Planner

In this section, we compare the performance of a conjunctive conformant planner against a number of conformant planners. The insight for the conjunctive planner comes from the fact that in many conformant planning problems, the plan that reaches the goal when any of a set of worlds may be true is the same as a plan that reaches the goal when the conditions of all of the worlds are true simultaneously. Intuitively, since the planner does not know

$P - T$	Conjunctive 733MHz			Probing 733MHz	GPT 850MHz	CMBP 850MHz	Cplan 850MHz
	Build Plan graph	SAT	Total	Total	Total	Total	Total
10-1	0.12	0.08	0.20	3.54	2.67	1.55	TIME
20-1	0.70	0.67	1.37				
30-1	3.59	5.56	9.15				
40-1	18.8	21.84	40.64				

Figure 8.24: Performance of the Conjunctive Planner

which world will be in force when the plan is executed, its plan is a union of plans that work in each world. This is often but not always the same as a plan that works in the union of the worlds. The proposed technique is thus to very quickly find a plan that works in the union of the possible initial conditions of the conformant planning problem. If such a plan can be found, we can then test it in each world separately to ensure it is conformant. If not, we can employ a conformant planner.

In the bomb in the toilet problem, the plan that that defuses the bomb if there is a bomb in any one unknown toilet is the same as the plan that defuses all bombs if there is a bomb in each toilet. Figure 8.24 illustrates the relative speeds of solving the STRIPS planning problem of defusing a bomb in each toilet versus the conformant planning problem of defusing a bomb in one of the toilets. The same plan found in each case. Clearly if we were able to find a conformant plan by converting the problem to planning in a single world, we would realize a significant time savings. If not, we would have wasted relatively little time when compared to the expected time to employ a conformant planner.

In practice, the success of this technique depends upon the planning domain. First, the planning domain may contain some resource that allows a plan to succeed if at most one world is true. For example, suppose we may dunk as many empty packages into the toilet as we like as long as we flush after each, but as soon as we dunk a package with a bomb, the toilet breaks. There is a conformant plan for this variation of the bomb in the toilet problem, but no conjunctive plan. Second, we can construct examples where planning in the conjunction of worlds actually produces plans that are simpler than the conformant plan, because the conditions of two worlds when taken together provide a simple path to

the goal. Since we cannot truly count on these two worlds being true at once, this is not a valid conformant plan. Third, when we assert the conditions for two possible worlds simultaneously, the propositional formula representing the planning problem may become inconsistent. In this cases, we can attempt to find conjunctive plans for subsets of the worlds taken together. If we are successful, these partially conformant plans can be used as plan fragments to be sewn together as in our fragment-based planner. We have not yet implemented this idea.

## Chapter 9

# Safe, Conformant Planning with Optimization

### 9.1 Motivation

In the previous chapter, we considered conformant planning, a generalization of deterministic planning wherein the initial state of the system is not known exactly. Given a description of the allowed operation of a physical system, a set of possible worlds, and a set of goals, a conformant planner returns a plan that achieves the goals in any world, if such a plan exists. Our experience suggests that conformant planning is too restrictive for the type of robust, autonomous operations we seek in spacecraft and other real-world systems. We have identified three main difficulties with attempting to generate conformant plans for these types of domains. First, the operations a candidate plan performs on the physical system cannot easily be categorized as allowed or disallowed. Each plan that reaches the goal has a different degrees of desirability, which is often determined by the existence of other plans. For example, spacecraft engineers would rather send a reset signal to a balky electronic component than power cycle it, as there is no guarantee the power will come back on once turned off. If the reset signal is not available, then a plan to power cycle the device is highly desirable if the device is necessary to preserve the operation of the spacecraft. Intuitively, we require some notion of safety. Safe plans are always desirable, and often dominate. However, critical goals may force the use of unsafe actions, and after a failure occurs, no safe actions may exist. Second, not all goals may be achievable in all worlds. As a degenerate example, if the spacecraft's camera is broken, then goals involving the camera are not achievable. If one possible diagnosis of the spacecraft includes a failure

of the camera but others do not, then goals involving the camera will be achievable in some worlds but not others. Finally, when we use planning in a control application, we may have a deadline by which the system must act. External events may further cut short the time during which the system can deliberate upon which sequence of actions to undertake. Thus even if a plan that safely achieves all goals in all worlds exist, we may not be able to generate such a plan before we must act.

Since we are interested in autonomous systems, we require a way to automatically reduce the scope of the planning problem when a conformant plan that achieves all goals in all worlds cannot be found in the available time. In addition, we need to ensure that some valid plan of action exists when we must act. We will need to balance safety, the goals that are achieved, and the number of worlds in which the goals that are achieved against the time required to find a plan. We define a planning domain in terms of three sets of constraints; the possible worlds, the goal set, and the safety constraints. Our planning problem is now to find the best plan, in terms of the constraints satisfied, before planning time expires.

## **9.2 Optimally Safe Conformant Planning**

In this section, we further motivate our intuitions about how and why to extend conformant planning with *safety* and *optimality*.

### **9.2.1 Safety**

STRIPS operators and a set of achievement goals that must be true at the end of a plan do not capture all constraints we would like to place on a satisfactory plan. We also would like to express goals of maintenance that constrain the states a plan may enter at any point in its execution. These additional constraints have three motivations; capturing exceptional conditions, capturing risk, and capturing the desire to preserve future capability. First, while our operators describe the general operation of the system, there are often specific exceptions. Some exceptions result from detailed analysis by spacecraft engineers. For example, if the spacecraft switches from the primary radio transmitter to the backup while the amplifier is powered, the predicted result is an electrical transient that would destroy the amplifier. Other exceptions might be empirical, based upon known, undesirable system

responses to certain command sequences. Regardless of their source, all such exceptions are encoded as a set of flight rules that human spacecraft operators use to check their hand-generated command sequences for safety. Since it would be difficult to capture and validate a first-principles model for these exceptions, we would like the flexibility to simply add these flight rules to our domain.

Second, our STRIPS operators capture a deterministic model of the system. We plan, execute the plan, then deal with failures on the next planning attempt. However, specific system configurations may increase the risk of undesirable and irrecoverable events occurring and are prohibited by flight rules. Again, rather than attempt to augment our model to include every possible contingency, we would simply like to explicitly encode states that are disallowed due to risk.

Finally, when fielding a planner for such an application, it's important to note that while the planner is reasoning only about the current planning problem, it will be called multiple times to generate plans that reconfigure the spacecraft. While we cannot know what the next reconfiguration request will be, and thus cannot plan for it, we would at least like to minimize the use of actions during the current plan that reduce our options for responding to future planning problems. For example, spacecraft typically employ pyrotechnic devices that can be used once for tasks such as opening the valves to a backup engine. Firing the pyrotechnic valves will abandon the primary engine and commit the system to using the backup engine. This reduces future flexibility and should be avoided. These examples place constraints on the states the system may enter during the execution of a plan. These constraints would be rather awkward to encode as STRIPS goals of achievement or in the preconditions of our operators. However, they are easily encodable as a set of maintenance goals that must be maintained at all points in time across multiple worlds. We call this set of goals the *safety constraints*. We refer to a plan that does not violate any safety constraints as a *safe plan*.

- Let  $\Psi$  be a set of predicates  $\{\psi_0 \dots \psi_n\}$  upon assignments to  $\mathcal{V}$ .  $\Psi$  is the set of safety constraints.

Ideally we would like to define a safe plan as one whose execution does not violate any

safety constraints. A slightly more complex definition is required, as failures of the system may cause one or more initial states to violate a safety constraint in a way that cannot be addressed by any action available to the planner. In this case, the execution of any plan violates a safety constraint. We therefore define a safe plan as one that does not increase safety violations from step to step. In the case where no initial states violate a safety constraint, the definitions are equivalent. Below we introduce the more complex definition of safety, as well as the definition of goal achievement and conformancy.

**Definition 9** Given a start state  $s$ , a plan  $p$  is **safe** with respect to  $s$  if each step in the execution of  $p$  starting at  $s$  does not violate any safety constraints not violated by the previous step. That is,  $\forall \psi_i \in \Psi$  if  $s_j$  precedes  $s_k$  in the deterministic execution of  $p$  on  $s$ , then  $\psi_i(s_j) \implies \psi_i(s_k)$ . Let this be denoted by  $\Psi_p(s)$ .

### 9.2.2 Optimal Plans

Ideally, given a problem we would like to find a plan that is both safe and conformant, ensuring it will safely bring the system to a desirable state regardless of the initial conditions. In most interesting domains, the ideal of simultaneously achieving complete safety and complete conformancy will not be achievable. This may be because the conformancy, safety, or time constraints are too strict. Conformancy may be the culprit because real systems can fail, leaving no single plan that succeeds in all initial states. This complicates the notion of what it means for a plan to satisfy a goal. For example, if we cannot satisfy a goal from all initial states we might wish to create a plan that satisfies the goal in one or more initial states but doesn't enter any unsafe states regardless of the initial state. In this case our plan will either succeed or it will fail to satisfy the goal in a way that safely sheds light on the true state of the system. For example, if the fuel valve is stuck or the tank is empty, opening a backup valve might produce thrust in the former case and reveal the tank is empty in the latter. Safety may need to be compromised if system failures prevent the safe action sequence leading to a goal state from being used, but unsafe alternatives exist. If the goal is important enough, safety constraints may become secondary. We might then prefer a plan that achieves the goal regardless of the initial state, but that minimizes

safety violations. For example, getting a spacecraft into orbit around its target planet at the cost of minimally damaging some subsystem might be more desirable than having an undamaged spacecraft fly uselessly into deep space. Time can be the culprit in that even if a safe, conformant plan exists, we may be unable to find it quickly. In domains where plans are being generated on-line to control a physical system, we have only a fixed amount of computation available before we exhaust the time available for deliberation and must act. A plan that is safe and conformant but that is delivered too late may have far less value than an on-time plan that satisfies most constraints. A spacecraft approaching a planet for orbital insertion has a small time window within which to slow down and be captured by the planet's gravity. A plan for insertion delivered after the insertion window has no value. Here, planning time is a constant and planning computation, and hence plan difficulty, must be adjusted to compensate.

These issues raise the question of how to plan if no safe, conformant plan can be found. Given we have multiple goals, each of which may be reachable from some initial states and not others, and multiple safety constraints, and a limited amount of time for deliberation, we would like a more flexible definition of what it means to have an acceptable plan. In addition, we require flexibility in determining how to choose between alternative, non-conformant plans under different planning contexts. This can be achieved by introducing a notion of optimization. If we cannot find a plan that meets all constraints in the allotted time, we must choose one or more constraints to drop. Different circumstances would lead us to choose different constraints to drop. Sometimes there might exist multiple plans that exhibit complete safety, complete conformity with respect to some goals, achievement of all goals for some subset of the initial states, or partial success in each measure. Ideally, we would like to find plans that are *pareto optimal* with respect to the set of suspended safety, conformance and goal constraints.

**Definition 10** Given a plan  $p$  and a set of constraints  $C$  violated by  $p$ ,  $p$  is **pareto optimal** if there exists no plan  $p'$  that violates the set of constraints  $C'$ , where  $C' \subset C$ .

In other words, if a plan satisfies all constraints that  $p$  satisfies plus additional constraints, we prefer it. If each plan satisfies a constraint the other does not, neither is preferred. If



there a single globally optimal plan exists, then it is pareto optimal. Note however that we can test any plan for pareto optimality locally, by reinstating the constraints it drops one at a time and planning again. This insight will prove useful in formulating an algorithm.

In this section we have motivated why it's useful to add notions of safety and optimality to the conformant planning problem. In the introduction, we hinted that optimizing the set of constraints the planner solves can be thought of as automatically adjusting the scope of the planning problem. We therefore refer to the family of planning algorithms we now introduce as the Safe, Conformant, Optimizing Planning Engine, or *SCOPE*.

### 9.3 SCOPE algorithms

Figure 9.1 illustrates the outline of a planner for finding pareto optimal plans. We have a set of goals,  $G$  to be achieved. We have a set of possible initial states of the world,  $W$ . We seek a plan that achieves all goals in  $G$  regardless of which world in  $W$  is the initial world. We have a set of goals of maintenance, called safety constraints,  $S$ , that a plan cannot violate when executed from any initial world in  $W$ . Intuitively, the safety constraints are a compact way of ruling out plans that enter states that unduly sacrifice future capability or undertake excessive risk. Ideally, we would like to find a plan that meets all goals of achievement  $G$  regardless of which world in  $W$  is the initial state of the system, without violating any safety constraints in  $S$ . Unfortunately, such a plan may not exist due to system failures, or we may not be able to find such a plan in the time allotted for planning. In this case, we'd like to find the best plan possible given the amount of time remaining for planning. The algorithm selects an initial set of constraints, that is a subset of the goals, worlds and safety constraints. For a fraction of the total time allocated for planning, it attempts to find a safe conformant plan for this problem. If a plan is found, the planner has a minimal set of constraints for which it can find a plan. It then selects additional constraints (goals, worlds and safety) and attempts to find a plan for the larger constraint set. If a plan cannot be found, the planner selects constraints to remove from the problem and attempts to find a plan for the smaller constraint set.

In order to instantiate this framework into a concrete planning algorithm, we must specify a policy for choosing which constraints to add or remove from the planning problem and

```

proc conceptualPlanner {
  Select initial constraint set from GUSUW
  while (time remaining) {
    for some time
      attempt to find safe, conformant plan given constraints
    if (a plan is found)
      Select additional constraints to add
    else
      Select constraints to suspend
  }
}

```

Figure 9.1: The Outline of a SCOPE Planning Algorithm

and a policy for determining how much time we will devote to each conformant planning attempt. In the next two sections, we introduce a variety of policies and describe the type of planner that results. We begin with simple policies based upon a user-specified preference function between constraints, then introduce policies that attempt to remove difficult constraints from the constraint set in order to maximize the likelihood of finding a plan in the allocated time.

#### 9.4 Partial Ordering of Constraints

A plan  $P$  is pareto-optimal if no plan exists that satisfies all constraints  $P$  satisfies plus an additional constraint. Note that there may be many pareto-optimal plans for a given planning problem. Each pareto-optimal plan satisfies a different subset of the constraint set. The planning algorithm attempts to find a pareto-optimal plan by expanding or contracting the set of constraints for which it is currently planning. In practice, not all constraint sets, and thus not all pareto-optimal plans, may be equally preferred. The order in which we add or remove constraints to the set for which we are attempting to find a plan should therefore be biased. We introduce the bias by specifying a partial ordering on the constraint set. In this section, we introduce three simple policies that completely obey the partial ordering. Constraints are added and removed from the current constraint set so that if a constraint is in the set, all constraints that are preferred in the partial order are also included.

- **Anytime Planning**

Anytime algorithms have the property that they generate solutions of increasing quality as

time passes. An anytime SCOPE planner is shown in Figure 9.2. We achieve the anytime property by removing nearly all constraints on our first call to Select and allocating all time to the subsequent planning call. If this low quality solution is found quickly, we can begin restoring constraints and generate incrementally improving solutions. This has the advantage that if even if a low quality solution turns out to be difficult to find, we can spend our entire time looking for it rather than be caught with no plan. We expect this policy to have a disadvantage when a plan that satisfies a significant fraction of the constraints is available. In this case, the time spent conservatively generating plans for small constraint sets may not leave sufficient time to generate a plan for a larger constraint set.

```

proc addPlan(Domain, G, S, W, Order, Time) {
  AllConstraints = GUSUW
  CSet =  $\emptyset$ 
  while (Time  $\neq$  0) {
    t = allocateTime(Time, CSet)
    C = Select(AllConstraints, Order)
    NewPlan = conformantPlan(t, Domain, CSet+C)
    Time = Time - t

    if (NewPlan  $\neq \emptyset$ ) {
      report NewPlan
      CSet = CSet + C
    }
  }
}

```

Figure 9.2: addPlan

- **Optimistic Search**

The counterpart to the anytime planning is an optimistic search, illustrated in Figure 9.3. We implement this policy by selecting all constraints on our first call to Select, and allocating a fraction of the available time to each planning attempt. If the planning attempt fails, constraints are removed. If no plan is found, constraints can be dropped more rapidly. We expect this policy will have an advantage when a plan exists for a significant fraction of the constraints. In this case, a significant portion of the planning time may be expended searching for this plan. The disadvantage is that for problems where only small subsets of the constraints are satisfiable, no plan may have been found when the

time allocated for planning expires.

```

proc DropPlan(Domain, G, S, W, Order, Time) {
  CSet =  $G \cup S \cup W$ 

  while (Time  $\neq$  0) {
    t = allocateTime(Time, CSet)
    NewPlan = conformantPlan(t, Domain, CSet)
    Time = Time - t

    if (NewPlan =  $\emptyset$ ) {
      /* If no plan relax the problem and try again*/
      CSet = CSet - Select(CSet, Order, Time, Failures)
    }
  }
  report NewPlan;
}

```

Figure 9.3: dropPlan

#### • Binary Search

This policy attempts to combine the best of anytime planning and optimism. For some fraction  $F$  of the initially available time, we attempt to solve the full constraint set. This may return a high quality plan. If  $F$  expires without generation of a plan, we suspend all constraints except a minimal subset, and allocate all remaining time to finding a minimal quality plan. If such a plan is found, we split the distance, in terms of number of constraints satisfied, between the largest constraint set for which we have a plan and the smallest constraint set for which we failed to find a plan. This process continues until time expires.

### 9.5 Optimization Via Relaxing The Problem Scope

The simple policies of the last section exhibit two potential weaknesses. First, by completely obeying the partial order supplied by the user, they risk generating no plan. That is, if the highest ordered constraint is simply not solvable, none of the algorithms given above will produce a plan. This may be appropriate in some domains, where the partial ordering is meant to specify that no constraint is useful if the highest ordered constraint cannot be solved. However, in many domains we may wish to disobey the partial ordering if certain

highly ordered constraints rule out all plans, or simply render the planning problem significantly more difficult. Second, when adding or dropping constraints, we are attempting to solve sets of highly related planning problems. In the algorithms above, no information is shared between planning attempts. Intuitively, we wish to add constraints when planning on the current constraint set has produced a plan. This plan is potentially a useful starting point for producing a plan for the augmented constraint set. We wish to remove constraints when the planning attempt on the current constraint set fails. The failure of the planning attempts may provide information about which problematic constraints should be removed in order to maximize the likelihood of generating a plan.

Conceptually, we can view the task of finding a plan that is pareto optimal in terms of the suspended constraints as two nested searches. First, we must search in the power set of the constraints to determine which subset of the constraints will be suspended. The active constraints define a planning problem and a space of plans. Second, we must search in this space of plans for a plan that is safe and conformant with respect to the remaining constraints. The fundamental insight is that the inner search, in addition to eventually providing a plan, also learns heuristic information for the outer search. Intuitively, if we cannot easily find a plan then there is a knot in the current constraint set that we cannot easily untangle. Examining the way in which the current constraint set thwarted our attempts to find a plan informs us as to which constraints should be suspended in order to ease the planning problem. Figure 9.4 illustrates this algorithmic framework. We attempt to find a plan, and in doing so estimate the relative amounts of problem difficulty being contributed by each goal, safety or initial state constraint. We record this estimate in the array variable *Difficulty*. If we cannot find a plan given the current constraint set, we select a constraint to suspend using *Difficulty* as our guide.

- **Optimistic Search with Learning**

The counterpart to the anytime planning policy is optimistic search with learning. We implement this policy by selecting all constraints on our first call to Select, and allocating a fraction of the available time to each planning attempt. If the planning attempt fails, we can examine the failure to determine which constraints are involved. These constraints

```

proc SCOPE_Planner {
  Select initial constraint set from GUSUW

  while (time remaining) {

    for some time
      attempt to find safe, conformant plan given constraints

    if (a plan is found)
      Select constraints to add biased by Order, Difficulty
    else {
      foreach plan thwarted by Constraint
        Difficulty[Constraint]++
      Select constraints to suspend biased by Order, Difficulty
    }

  }
}

```

Figure 9.4: A More Complete Planning Framework

are removed. If a plan is found, the failure information can be used to estimate which constraints could be added to the problem and still allow a plan to be found. If no plan is found, constraints can be dropped more rapidly. This policy has the advantage that some estimate of the difficulty of finding a plan is used in addition to utility in determining the constraint set. A plan for a constraint set must be found in order for its utility to be relevant. The disadvantage is that no plan may have been found when the time allocated for planning expires.

- **Reversal of Fortune**

This policy attempts to combine the best of learning the difficulty of constraints and anytime planning. For some fraction  $F$  of the initially available time, we attempt to solve the full constraint set, or a relaxation. This may return a relatively high quality plan, and will provide some information about the relative failure rates of constraints. After  $F$  has expired, we suspend almost all constraints except for those known to be of low difficulty and allocate the full  $Time-F$  period to finding a low quality plan. Once this low quality plan is found, we incrementally improve upon it by restoring non-difficult constraints as time allows.

- **Anytime Planning with Fragments**

As with the simple anytime planner, anytime planning with fragments starts with a minimal constraint set and adds constraints as long as a plan is found. In this variation, we use the plan from the previous constraint set as a fragment to seed the fragment-based conformant planner. As discussed in the thesis, there is no guarantee that a plan for a subset of a set of constraints is a sub-plan of a plan for the entire set. However, attempting to add actions to an existing plan to satisfy the expanded constraint set is a reasonable place to start.

### 9.5.1 Computing *Difficulty* for Goals and Safety

In addition to updating *Difficulty*[ $w$ ] for each world in the plan algorithm, we must update *Difficulty* for each goal and safety constraint during our attempts to find plans for a single world. A single world planning attempt corresponds to a call to *satisfy* in order to find a satisfying assignment to our propositional formulation of the planning problem. Our SAT solver is a slightly modified version of *Satz\_rand* (Kautz & Selman 1999), which is itself a randomized version of *Satz* (Li & Anbulagan 1997). *Satz* is a Davis-Putnam-Logemann-Loveland (DPLL) procedure (D. Putnam & R. Davis 1960) that makes use of unit propagation to find a satisfying assignment of a propositional formula. Our only modification to *Satz\_rand* is to the unit propagation step, as illustrated in Figure 9.5. When unit propagation causes a disjunction to become a unit clause, it constrains the remaining variable to take on a truth value. This clause is recorded as the *support* of the variable's value. Then, when we find an inconsistency, we can quickly trace back through the support of the variables involved in the inconsistency and increase the *Difficulty* of each clause involved in constraining those values. We are interested in the difficulty of constraints from the planning domain, whereas each clause is a part of a model of the planning domain unrolled over a finite number of time steps. To map between the two, we simply record which clauses result from each domain constraint when the domain is compiled to the propositional representation.

```

proc Propagate(Formula,Difficulty) {
  /* Simply unit propagation with recording of support*/
  while (empty clause  $\notin$  Formula  $\wedge$  a unit clause U exists) {
    assign variable u in U to satisfy U
    support[u]=U
    Formula = Formula - U
    for all Clause  $\in$  Formula that contains u {
      if (Clause agrees with u)
        /* Clause is satisfied by u*/
        Formula = Formula - Clause
      else
        /* u cannot satisfy Clause*/
        Clause = Clause - u
    }
  }

  /* If assignment is inconsistent, increment difficulty of */
  /* every clause involved in producing the inconsistency*/
  if (empty clause  $\in$  Formula) {
    find support of all variables originally in clause
    Difficulty[support]++
  }
}

```

Figure 9.5: Unit Propagation With Support Tracking



## **Chapter 10**

# **Experimental Results for SCOPE**

### **10.1 Introduction**

In this chapter, we examine the behavior of several simple control schemes for SCOPE. In the first section, we attempt to develop a simple intuition about the interaction between a control scheme, the complexity of the problem being solved and the amount of time allocated for planning. In the second section, we make more concrete observations about the behavior of SCOPE and fragPlan for problems where a conformant plan exists. In the third section, we consider the behavior of SCOPE when there is no plan that satisfies the set of constraints that make up the planning problem. In this case, SCOPE must choose a subset of the constraints and return a plan for this subset.

### **10.2 Overview of SCOPE Results**

Recall that given a set of worlds and constraints that define a planning problem, the SCOPE algorithm makes multiple attempts to find a conformant plan on different subsets of the worlds and constraints. Before discussing strategies for choosing these subsets, we develop a simple intuition of the challenge these strategies must address. Consider the case where we have a conformant planning problem, and that given sufficient time, the problem is solvable. That is, there are enough time steps and resources in the problem to allow a conformant plan. Suppose we may suspend any number of the constraints. Figure 10.2 is a caricature of the relationship between the number of constraints active in the problem

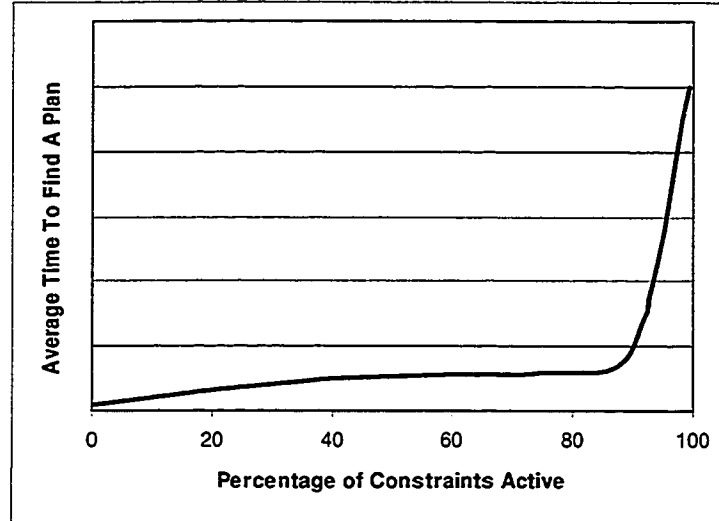


Figure 10.1: An Intuition of Planning Time Vs. Constraints

versus the effort required to solve the problem. The horizontal axis represents what percentage of the constraints defined in the original problem that remain unsuspended. The vertical axis represents the difficulty of finding a plan for the unsuspended constraints. For example, recall the BTC problem. Our experience is that when there are exactly enough time steps to solve the problem for  $N$  packages, finding a conformant plan for  $N$  packages has low to moderate difficulty. However as soon as allow SCOPE to discount a few of the worlds (*i.e.*, packages that might have the bomb) the problem quickly becomes significantly easier to solve. Similarly, we have created conformant logistics problems where it is quite a challenge for fragPlan to find a plan that delivers a package to each of  $N$  cities over mined routes. However, as we allow SCOPE to abandon delivering a few of the packages, the problem obviously becomes easier to solve<sup>1</sup>

The exact shape of the difficulty versus constraints curve for these two problems differs. However, the basic intuition we are attempting to convey is that for each problem domain and problem instance, there is some relationship between the constraints that are included and the average time required to find a plan. Imagine we have an oracle that will report

<sup>1</sup>We may of course encounter problems where some inclusion of some constraints creates significant more difficulty than others. In that case, this general curve would only hold if we were able to add constraints to the problem in order of by their difficulty. For the moment, we ignore this issue for the purposes of discussion.

Control scheme	Initial constraints	Allocation of $t$ seconds	Action on Success	Action on Failure
conformant planning	all	$t$	Return plan	No plan
addPlan	1	$t$	Add constraint	Return last plan
dropPlan	all	$t/k$	Return plan	Drop constraint
binPlan	all/2	$t/k$	Add $1/i$ constraints	Drop $1/j$ constraints

Figure 10.2: Four control schemes for SCOPE

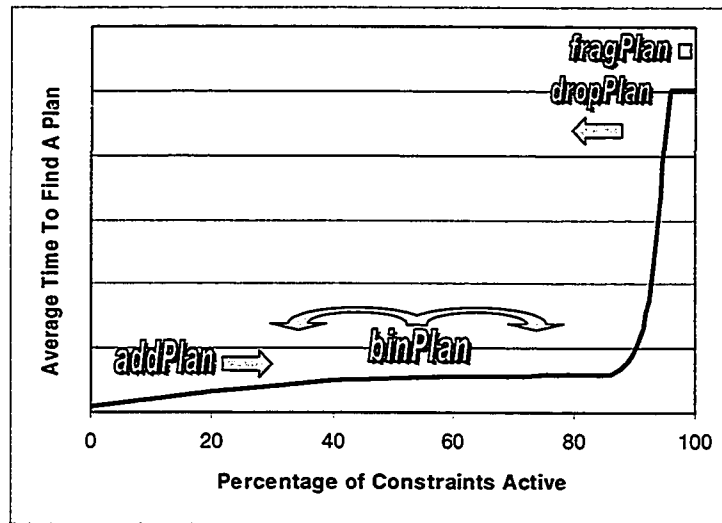


Figure 10.3: Four Simple SCOPE Strategies

number of constraints  $n$  (or more generally, a specific set of constraints) of the current planning problem that can be solved given some amount of time  $t$  for planning. Given this oracle, we would be able apply all of the available time  $t$  to finding a plan for the maximum set of constraints that can be solved in time  $t$ . Given we have no such an oracle, we must take an educated guess at  $n$ . We may overshoot it, and waste valuable time failing to find a plan, or undershoot it and waste valuable time generating a plan for a few constraints when far better ones were achievable. In either case, we must then take an educated guess at which constraints to attempt given the time remaining. This is the basic schema for a SCOPE algorithm: make a guess at which constraints can be solved, attempt to solve the problem, examine the result and determine which set to attempt next in the time remaining.

In order to instantiate a SCOPE algorithm, we must specify which set of constraints

will define the initial planning problem, how much time will be expended on each planning attempt, and how the constraint set will be adjusted after each attempt. Figure 10.2 captures these specifications for four simple control schemes for SCOPE, while Figure 10.3 is a visual representation of how each moves in the space of constraints. The second column of Figure 10.2 describes the set of constraints that are attempted in the first planning attempt. The third captures the fraction of time allocated to the current planning attempt, given that there are  $t$  seconds remaining to make planning attempts. The fourth column describes what happens if the current planning attempt succeeds and returns a plan. The fifth column describes what happens if the current planning attempt fails and no plan is returned.

- **Conformant Planning** is a special case of SCOPE where our initial planning attempt is to solve a conformant planning problem with all constraints, using all available time. If this attempt is successful, we return the plan that was generated. If no plan is found on the first attempt, the algorithm returns no plan, as all of the time available for planning has been exhausted. Given we are using the fragPlan algorithm of Chapter 7 as the conformant planner in SCOPE, this SCOPE strategy is equivalent to fragPlan.
- **addPlan** is an anytime implementation of SCOPE. Its focus is to ensure some plan exists when the time allocated for planning is exhausted. As such, it begins by attempting a planning problem with only one constraint active. It allocates all available time to this planning problem. Intuitively, if we cannot find a plan for the simplest possible problem, we should not attempt more complex problems<sup>2</sup>. If the planning attempt is successful, we select a one or more constraints to add to the problem based upon a partial ordering, and plan again. The plan generated for the current constraint set may be used as an initial solution for the planning problem with an additional constraint. The next planning attempt may then augment the existing plan or quickly discover that it has to discard it and begin from scratch. The last plan successfully generated is returned when time expires or when we have found a plan for all constraints.
- **dropPlan** attempts to find a conformant plan, but leaves time in reserve for additional

---

<sup>2</sup>We could also hold a fraction of the available time in reserve to guard against the case where the first constraint we attempt to plan for is not solvable.

planning in case no conformant plan can be found. We have expressed the amount of time that is allocated to the current planning attempt as  $t/k$ . In the majority of our experiments,  $k = 2$ . That is, we first attempt to find a plan for all constraints in time  $t/2$ . If that fails,  $t/2$  remains, and  $t/4$  is allocated to the next attempt. A more complex or adaptive schedule could be used to determine how quickly to give up on the current planning problem. If the current attempt fails, one or more constraints are dropped from the problem, and another planning attempt is made. Again, a variety of techniques could be used for scheduling which and how many constraints should be eliminated upon failure. We may simply follow a partial order supplied by the user. We may attempt to eliminate the most troublesome constraints first. To this end, during each conformant planning attempt we can record how frequently each constraint caused a potential plans to be eliminated. This gives us a heuristic estimate of the contribution of each constraint to the overall difficulty of the planning problem. We might also obeying the partial order supplied by the user and eliminating constraints that appear to be difficult first. For the experiments reported here, we removed one constraint per time step, and followed the partial order supplied by the user.

- **binPlan** attempts to balance the characteristics of `addPlan` and `dropPlan` via a binary search for  $n$ , the correct number of constraints to attempt given the time available,  $t$ . Intuitively, when  $n$  is high `addPlan` may spend a significant amount of time finding a number of plans that satisfy fewer constraints than  $n$ . It may therefore never attempt to solve a problem with  $n$  constraints before time expires. Similarly, when  $n$  is low, `dropPlan` may spend a significant amount of time attempting to find plans that satisfy more than  $n$  constraints and failing. By the time it attempts to solve  $n$  constraints, it may have insufficient time remaining to find a plan. The intuition behind `binPlan` is that it seeks to grow its constraint set faster than `addPlan` when the problem is easy, and drop constraints faster than `binPlan` when the problem is difficult. It begins by attempting to solve half of the constraints of the problem instance, again using  $t/k$  or some other scheduling of the time remaining. If this attempt is successful, `binPlan` adds to its constraint set  $1/i$  of the difference between the constraint set it just satisfied and the full constraint set. If the

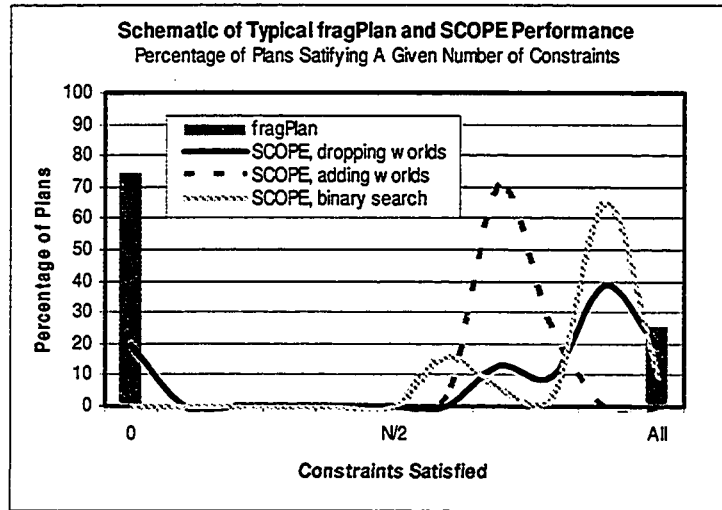


Figure 10.4: Typical Performance for Four Simple SCOPE Strategies

attempt fails, binPlan discards  $i/j$  of the currently active constraints. In our experiments,  $i = j = 2$ .

Before discussing details experimental results, we next give an intuition of the performance of planners driven by each of these control strategies relative to one another. Figure 10.4 illustrates one example of the relative performance of fragPlan, addPlan, dropPlan and binPlan on a single problem instance. Each planner was run for 30 trials of the same problem instance. The graph illustrates how many of the constraints were satisfied by the plan returned by each trial. The horizontal axis is the number of constraints that were satisfied by each plan. The vertical axis is the percentage of plans solving that number of constraints. Thus, 25% of the attempts by fragPlan resulted in plans that satisfied all constraints, and 75% resulted in zero constraints being satisfied because no plan was found. This is a BTC instance, but results are meant to be representative for the purposes of discussion. We next provide an intuition as to why these graphs have these shapes.

Note that fragPlan always finds the largest number of conformant plans, as it dedicates all of its time to attempting to solve all constraints. Of course, due to this go-for-broke approach fragPlan also experiences the maximum number of failures where no plan is returned at all. Note that addPlan forms a peak with a wide base much of the way up the

constraint set. Intuitively, in many domains we have examined, plans with very few constraints are quite easy to find. Each `addPlan` trial races across the horizontal axis, quickly knocking out plans that solve 1, 2, 3, . . . constraints. Even though `addPlan` will allow each planning attempt to use the entire time allocation  $t$  if needed, each planning of the first few planning attempts use only a small fraction of this time. Thus `addPlan` has a significant fraction of  $t$  remaining when it begins to attempt more difficult plans. However, as the constraint accrue, the time `addPlan` has expended finding plans begins to accumulate more quickly, and the planner runs out of time to find the next plan. If our conformant planner and underlying SAT procedure were deterministic, all trials would achieve the same number of constraints. We would then have a narrow peak where the wide peak of `addPlan` appears. Due to the use of randomized algorithms, some trials take longer than average on the simpler planning problems and fall short of the peak shown in the graph. Other trials are able to reach the peak quickly, and have a significant amount of time left. Some fraction of these are able to generate plans for one, two or more additional constraints.

Note that `dropPlan` finds the second largest number of conformant plans, after `fragPlan`. This is expected, as it applies the second largest amount of time,  $t/2$ , to attempting to find a plan for all constraints. Of course, a significant number of those attempts fail, and each failed attempt uses up its entire  $t/2$  allocation. Now `dropPlan` only allocates time  $t/4$  to solving the problem with one less constraint. Intuitively, `dropPlan` is in a race against time, using up  $t/2$ , then  $t/4$ , then  $t/8$  of its time allocation as the problems become easier due to elimination of constraints. There are peaks where the problem with  $c$  constraints removed becomes easy enough to solve with some degree of certainty in a time allocation of  $t/2^c$ . Unlike `addPlan`, in a significant fraction the trials `binPlan` finds no plan.

For this problem instance, `binPlan` does appear to combine the best of `addPlan` and `binPlan`. It first attempts to find a plan for  $1/2$  of the constraints in  $t/2$  time. The constraints for this problem are relatively loose, and in every trial a plan is found very quickly for this constraint set, just as in `addPlan`. Thus we have a plan for half the constraints in the bag, and much of  $t$  remains. Unlike `addPlan`, `binPlan` does not next attempt  $1/2 + 1$  of the constraints. It jumps directly to  $3/4$  of the constraints. A few trials are not able to find a plan

at  $3/4$  in half of the time remaining. These failures use up a significant amount of time, but unlike dropPlan, binPlan jumps all the way down to  $5/8$  of the constraints. All of the trials that were not successful at  $3/4$  are successful here. The majority of trials do succeed at  $3/4$  of the constraints. They then use half of the remaining time to find a plan for the full constraint set. A small percentage of them succeed, but the majority remain at  $3/4$ .

### 10.3 Observations on SCOPE and fragPlan

In this section, we present a set of observations about the performance of SCOPE on planning problems where a conformant plan exists. Since a conformant plan exists, we can compare the performance of SCOPE against fragPlan. Figures 10.5, 10.6, 10.7 and 10.8 illustrate the performance of SCOPE and fragPlan on three different conformant planning domains. Each figure is a histogram of the number of constraints satisfied for each of 30 plans for each planner variation, for a given amount of computation time. Figures 10.5 and 10.6 are for the BTC problem. Each world in this problem represents the possibility that the bomb is in a specific package. Each trial of fragPlan returns either a plan that satisfies all worlds (dunks the bomb no matter which package it occupies) or returns no plan. The SCOPE variations may choose not to satisfy all worlds in the problem. Figure 10.7 illustrates the modified RING domain. Recall that in this domain a robot is in a maze containing  $n$  rooms, and the goal is for window in each room to be locked. We specify this as a set of goals, each of which requires that one of the windows be locked. Here, fragPlan either returns a plan that locks all windows or returns no plan. Here SCOPE variations may choose not to satisfy all goals, and may leave some windows unlocked or open. Figure 10.8 is a logistics problem where a delivery truck must deliver a package to a set of cities connected to one another in a linear array. In each world, one route between adjoining cities may be mined. The problem is designed to be very tightly constrained, so even very small instances are difficult for a fragment-based conformant planner. All runs were performed 733MHz Pentium III PC with 256M of RAM, running Windows NT 4.0.

**Observation 10** *SCOPE pays a penalty in terms of conformant plans for not investing all of its time on the conformant problem.*



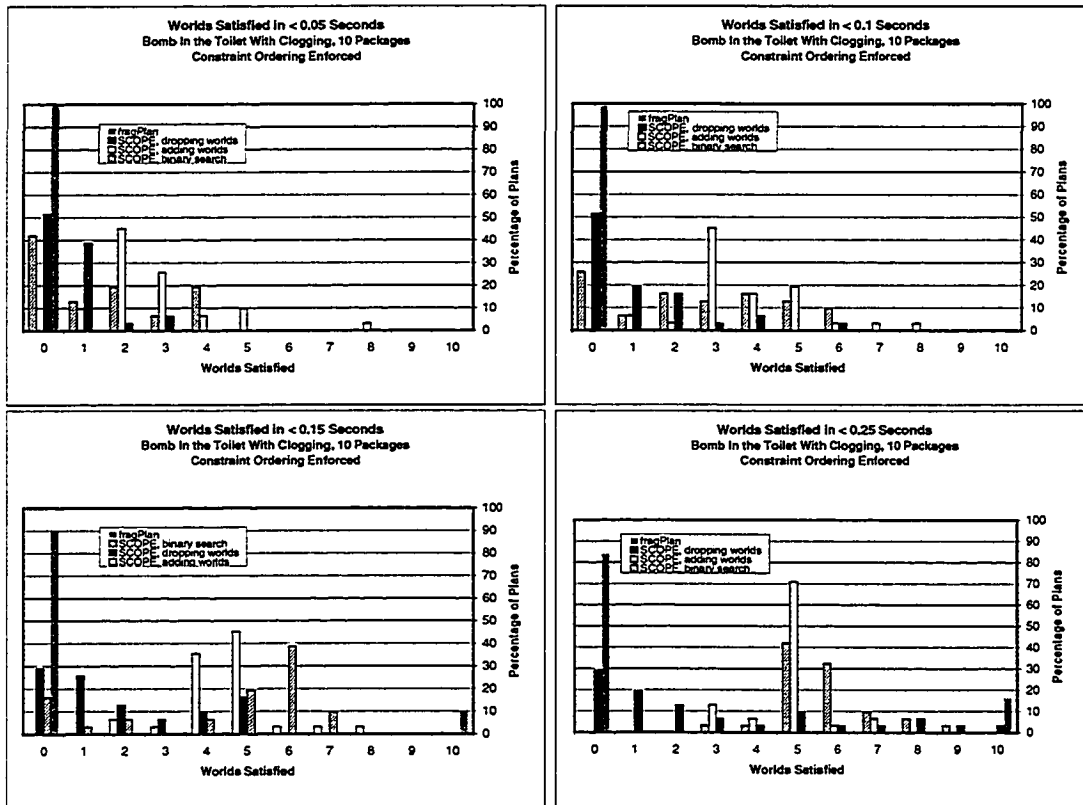


Figure 10.5: SCOPE on BTC, 10 Packages, 22 Time Steps, Part 1

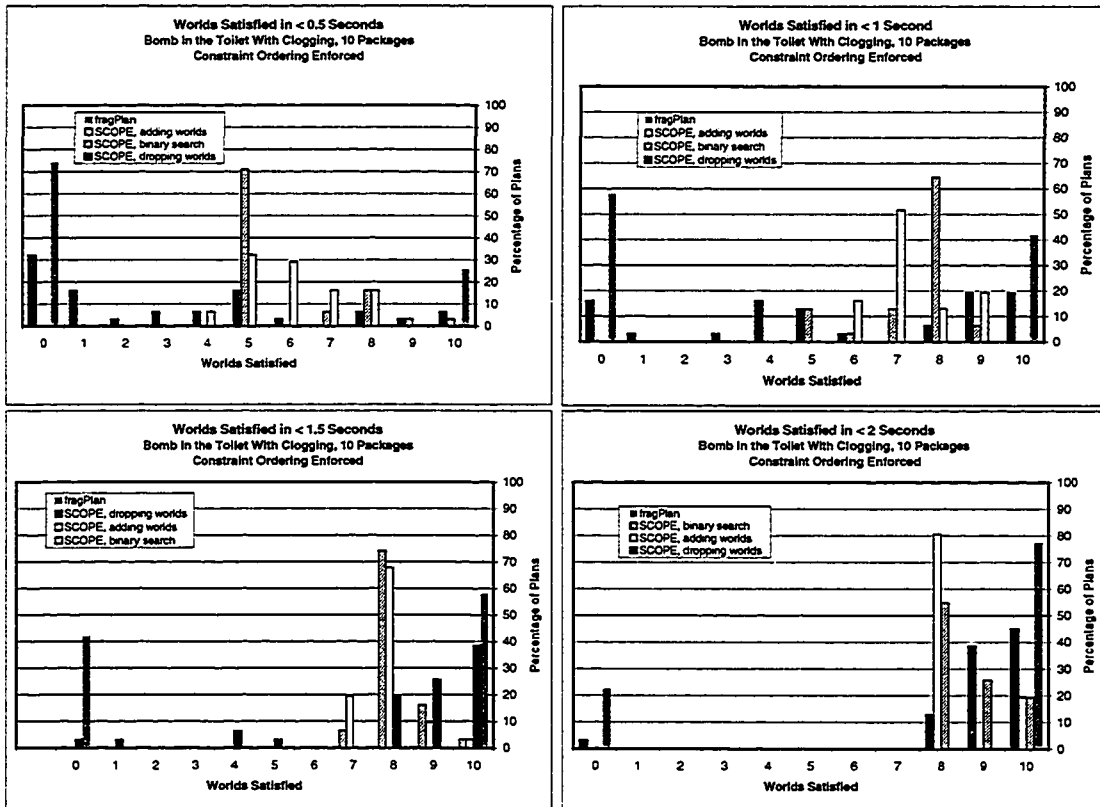


Figure 10.6: SCOPE on BTC, 10 Packages, 22 Time Steps, Part 2

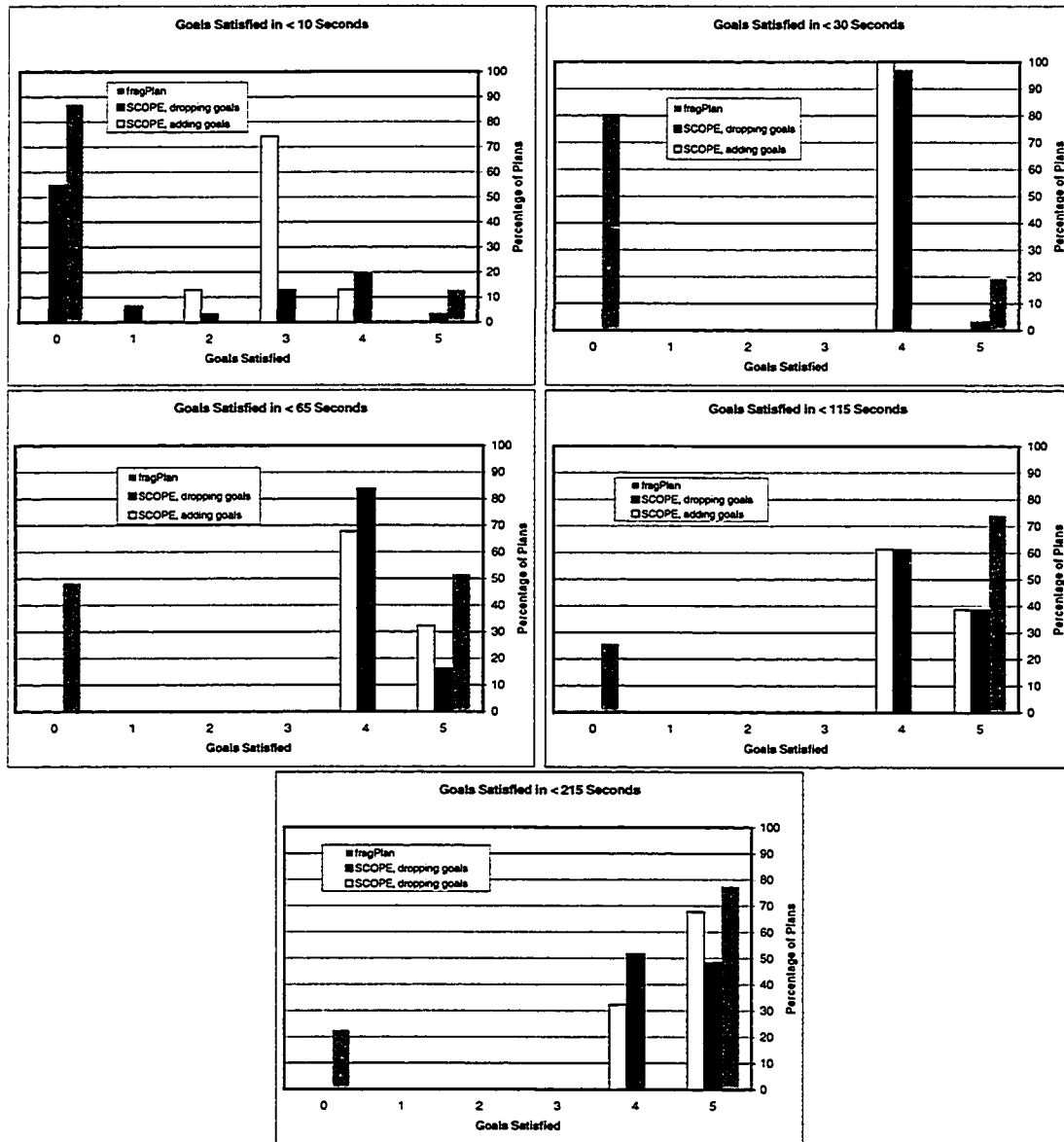


Figure 10.7: SCOPE on Modified Ring World of Size 5

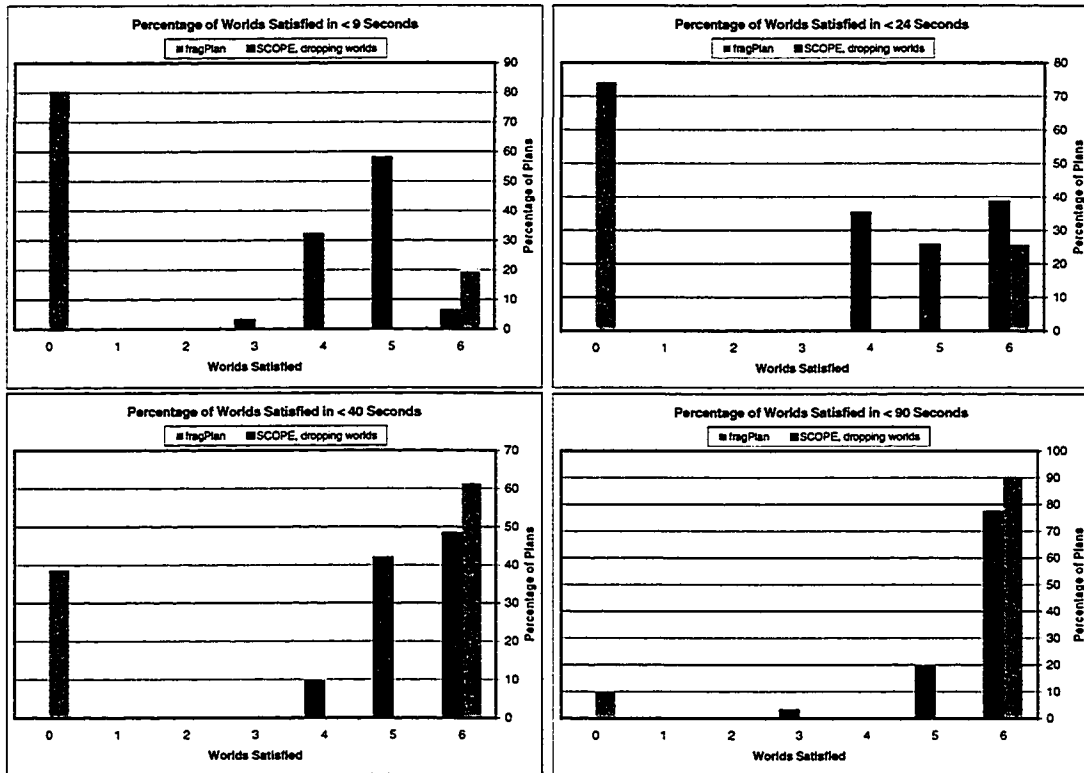


Figure 10.8: SCOPE on the Linear Logistics Problem of Length 6

It's clear that when we care only about finding a conformant plan for all constraints as often as possible, fragPlan is the planner of choice. None of the SCOPE planners can be expected to dominate fragPlan when judged only by the number of conformant plans returned. The dropPlan strategy invests only 1/2 of its time attempting to solve the full conformant planning problem. The addPlan and binPlan strategies attempt a number of plans before attempting the full conformant plan, and thus can never apply as much time as fragPlan to the problem of satisfying all constraints. Accordingly, in all of our experiments, no SCOPE algorithm ever generates more conformant plans for a reasonably sized set of trials than fragPlan. In addition, as time is increasingly limited the penalty paid by SCOPE for not investing all of its time in the full conformant problem is increasingly severe. For example, consider the 0.5 second case of Figure 10.6.

**Observation 11** *In terms of expected worlds solved or expected goals met, SCOPE performs better and is far less sensitive to the amount of time allocated than SCOPE.*

Figure 10.9 illustrates the trend of expected performance, in terms of the expected number of worlds or goals satisfied, versus the time allowed for computation. The expected performance is computed from the sets of 30 trials illustrated in Figures 10.5, 10.6, 10.7 and 10.8. The expected performance at each time allocation is found by computing the average number of worlds or goals satisfied over each set of 30 trials. Note that the all-or-nothing behavior of fragPlan has a significant impact on its expected performance. The expected performance of fragPlan is never higher than the SCOPE variations on any problem we have attempted.

**Observation 12** *SCOPE dominates unless plan utility is strongly biased to achieving all goals or satisfying all worlds.*

This is in some sense a corollary of the previous two observations. Consider for example the performance illustrated in the 9 second graph of Figure 10.8. Let  $U_N$  be the utility of a plan for N worlds or goals. The expected value of fragPlan is  $0.19U_6 + 0.81U_0$ . The expected value of dropPlan is  $0.06U_6 + 0.58U_5 + 0.32U_4 + 0.03U_3$ . Keeping in mind that we can reasonably expect that  $U_0 \leq 0$ , note that fragPlan can only dominate if  $U_6 > 4.62U_5 + 2.46U_4 + 0.23U_3 - 6.23U_0$ .

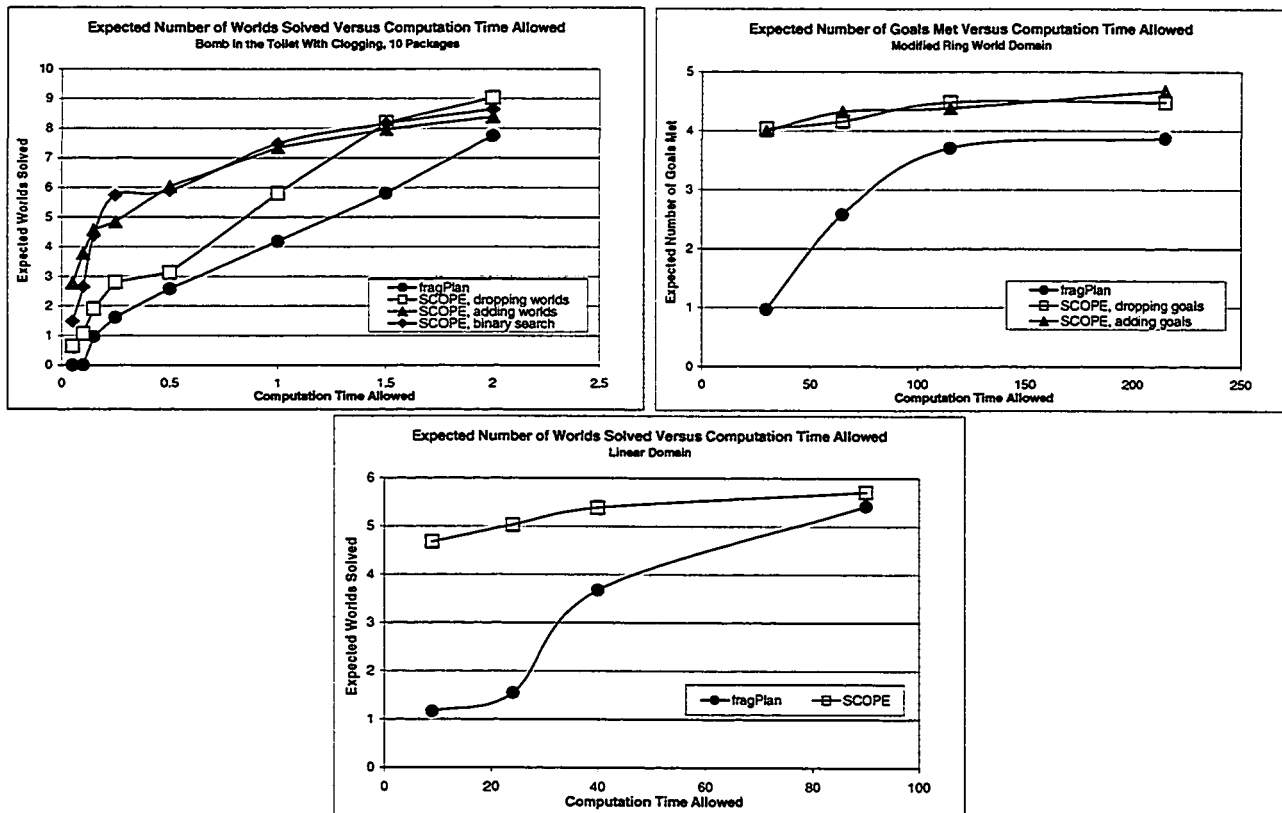


Figure 10.9: Expected Performance versus Time for fragPlan and SCOPE

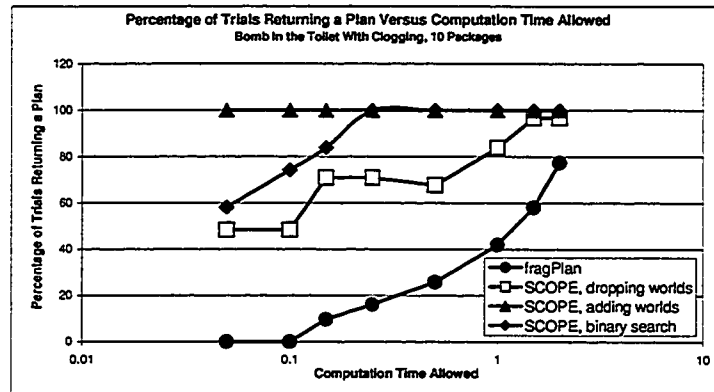


Figure 10.10: Plans found by SCOPE compared to those found by fragPlan

**Observation 13** *As computation time decreases, SCOPE has an increasing advantage in avoiding the situation where no plan is produced.*

In control domains, it may be critical to have some plan ready to execute when the time allocated for planning is exhausted. Figure 10.10 captures the percentage of trials that return a plan for each SCOPE strategy for the BTC trials shown in Figures 10.5 and 10.6. Each point represents the amount of time allocated to one of the strategies, on the horizontal axis, and the percentage of trials out of 31 that returned a plan, on the vertical axis. Note that the time axis is logarithmic simply for the purpose of separating the points for legibility. As expected, all SCOPE variations return a plan significantly more often than fragPlan.

**Observation 14** *When extremely little computation time is available, addPlan dominates, on the BTC problem.*

Consider the left of Figure 10.10, where  $t$  is quite small relative to the complexity of the problem. Here addPlan's strategy of applying all time to simple problems dominates when the metric of interest is the number of trials that successfully return a plan. Recall the top left graph of Figure 10.9. Here addPlan also dominates, in terms of the expected number of worlds satisfied per planning trial, at low time allocations. If we consider, for example, Figures 10.5 and 10.6 we note that at 0.05 seconds, only addPlan is generating any plans that satisfy more than 4 worlds.

**Observation 15** *Except under very low or high time allocations, binPlan does quite well*

*relative to the other strategies, on at least the BTC problem.*

Note that in the BTC problem, at the top left Figure 10.9, binPlan does quite well except in the case where the available execution time is quite small or fairly large. Recall that given  $t$  seconds for planning, addPlan begins by allocating  $t$  to the simplest planning problem it can construct, dropPlan begins by allocating  $t/2$  to the complete constraint set, and binPlan allocates  $t/2$  to solving the planning problem for half of the constraints. As noted above, addPlan does very well where the time allocated is very short. At the right end of the graph, as  $t/2$  approaches the time needed to find a conformant plan, dropPlan begins to dominate over binPlan's strategy of attempting half the constraints first. However, at no point does binPlan do significantly worse than the others, and it often dominates. Direct examination of the planning trials shown in Figures 10.5 and 10.6 is somewhat more telling than a graph of the expected performance. Starting at the graph of performance with 0.05 seconds allocated, all planners are generating plans centered at the left end (0 to 2 worlds satisfied) of the graph, with addPlan dominating. As the time allowed for computation is increased, the peak for each planner moves rightward, with binPlan moving most quickly. Only as we approach 1.5 seconds per trial does dropPlan have a significant jump leftward. The expected values for addPlan, binPlan and dropPlan are similar at 1.5 and 2 seconds. However, this is somewhat misleading. Consider Figure 10.6. At 1.5 and 2 seconds dropPlan generates a number of plans that satisfy one more world than binPlan, while binPlan bests many of the plans of addPlan by one world.

#### **10.4 SCOPE Performance When No Conformant Plan Exists**

In this subsection, we consider SCOPE performance when no conformant plan exists. First, we consider the case where some resource limitation allows us to create a plan that works in many worlds, but not all. Second, we consider the case wherein there is not enough time for acting (the planning horizon) to execute the minimum length conformant plan.

**Observation 16** *SCOPE effectively handles domains where no conformant plan exists because of resource limits.*



Sweepers/ Mines	0.15 Seconds/Run			1 Second/Run		
	Solved	Time	Calls	Solved	Time	Calls
3/6	3	0.14	73	3	1.07	608
4/6	4	0.14	54	4	1.01	432
5/6	4.87	0.13	43	4.97	0.90	297
6/6	6	0.04	9	6	0.04	7

Figure 10.11: SCOPE when resources set a maximum number of solvable worlds

Figure 10.11 illustrates runs from a logistics domain where packages must be delivered to a set of cities from a centralized warehouse. There are six worlds, each of which corresponds to the possible location of a mine that has been placed on one of the routes between cities. There is a minesweeper which removes a mine, if any, from a route. In order to limit resources, we have modified the domain to specify that once a minesweeper has been sent to a route, it cannot be used on another route, regardless of whether a mine was cleared. When  $N$  sweepers are available, any  $N$  worlds can be satisfied. The first column denotes the number of mine sweepers that are available. The remaining columns illustrate the number of worlds solved, on average over 31 addPlan runs, when 0.15 seconds and 1 second are allowed per run. Note that SCOPE quickly finds a plan for the maximum number of worlds that can be accommodated. It then spends all remaining time searching for a plan that improves upon what is in fact the best possible plan, since it cannot determine that there is no better plan.

**Observation 17** *SCOPE effectively handles the situation when the time available for acting (the planning horizon) is insufficient for a conformant plan.*

In addition to resource problems preventing the existence of a conformant plan, the number of actions the user allows the planner to consider (the planning horizon) may prevent the existence of a conformant plan. In this case, SCOPE either drops goals to eliminate the need for the actions that achieve those goals, or drops worlds to free the space the fragments those worlds would require. Figure 10.12 illustrates the performance of dropPlan on a difficult logistics problem and a BTC problem as the length of the planning horizon is modified. The dotted line represents the performance of an optimal plan. This line denotes the maximum possible number of worlds or goals that can be satisfied given the limitations

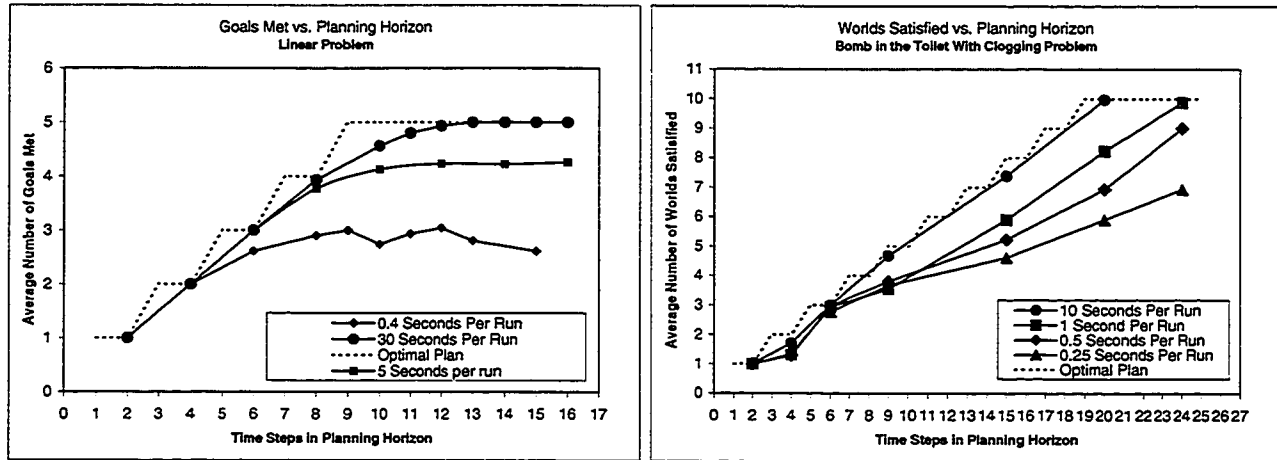


Figure 10.12: Performance of dropPlan versus horizon for several run times

of the planning horizon. The other lines represent the performance of dropPlan with different time allocations. Note that given sufficient time, dropPlan finds the optimal plan. As time is reduced, the amount of time dropPlan wastes attempting to solve the full constraint set negatively impacts its ability to find what is actually an optimal plan.

**Observation 18** *When few constraints can be satisfied and time is short, addPlan dominates.*

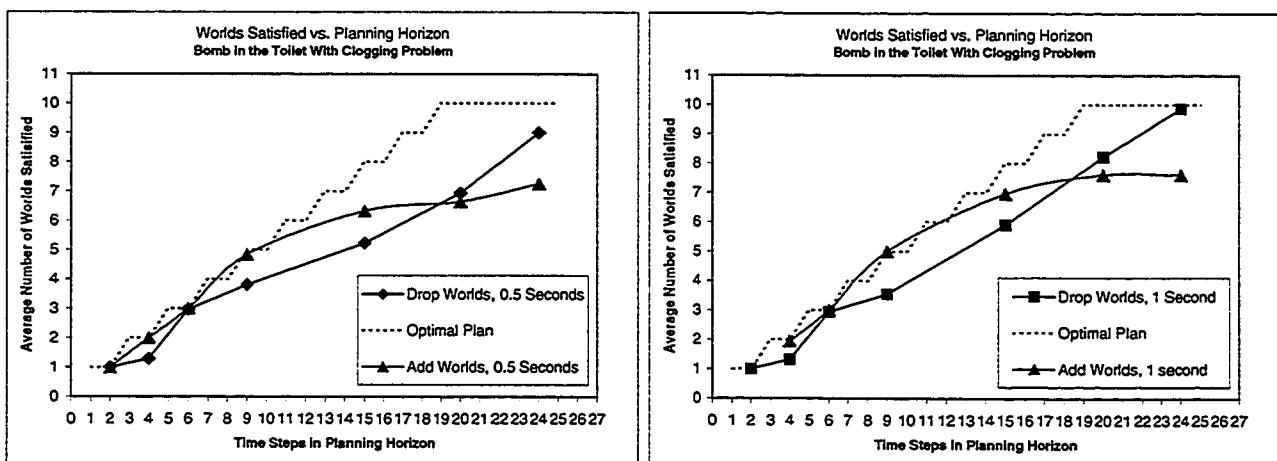


Figure 10.13: Performance of addPlan and dropPlan versus planning horizon

As one would expect, addPlan has an advantage when the planning horizon prevents the full set of constraints from being satisfied. Consider Figure 10.13. The addPlan strategy has the advantage until the lengthening horizon has made the problem relatively easy.

## Chapter 11

### Related Work in Acting Under Uncertainty

Recall that our discrete control problem can be modeled as a partially observable Markov decision process, or POMDP. A policy is a mapping from belief states to actions. Given any belief state, the policy specifies an action to be taken. To solve a POMDP is to create a policy that for any distribution over the state space returns the optimal action to take in order to maximize the expected reward given the partial observability. The details of finding an exact solution to a POMDP are beyond the scope of this proposal but a number of algorithms for finding such a solution exist (Sondik 1971; Cheng 1988; Littman, Cassandra, & Kaelbling 1995). Unfortunately, even the most efficient of these is not generally tractable on problems more than tens of states (Hauskrecht 2000; Kaelbling, Littman, & Cassandra 1998; Zhang, Lee, & Zhang 1999).

Fortunately, a wide variety of techniques for more limited versions of POMDP solutions have been developed. These techniques typically involve reducing the problem complexity by making simplifying assumption about uncertainty (*e.g.*, assuming the world is observable or has deterministic actions), generating something other than a policy that maximizes expected reward from any state (*e.g.*, assuming there is a single goal state that must be reached) or some combination. MDP techniques generate a full policy, applicable in any state, but assume the current state is always observable. Conformant planning techniques assume the initial state is unobservable but contained within a small set, and generate a plan that succeeds (possibly with some probability) regardless of the initial state. Contingent planning techniques generate a branching plan whose branches are chosen at execution

time based upon the results of observations. Belief replanning makes an initial assumption about the start state or initial distribution and creates a deterministic plan appropriate to that assumption. If at any point the belief state predicted by simulating the deterministic plan diverges significantly from the belief state resulting from execution of the plan, either the initial assumption was incorrect or the plan did not behave deterministically. In either case, a new plan is generated from the divergent belief state. The remainder of this chapter describes these methods and their applicability to the problem at hand.

### **11.1 Belief Replanning**

Belief replanning occurs at the opposite end of the uncertainty spectrum as an optimal policy. This technique is based upon the most likely state (or MLS) assumption, wherein the agent always assumes it is in the state with the highest likelihood. Belief replanning creates a deterministic plan from the current state to the goal state, and replans when it's clear something has not gone as expected. The algorithm starts by assuming it is in the most likely world state and that a deterministic idealization of the action model holds. In this idealization, the most likely outcome of an action in a state is assumed to be the outcome that always results from performing that action in that state. Given these assumptions, the algorithm generates a deterministic plan from the most likely state to the goal state along the appropriate deterministic actions. In addition, it generates a sequence of predicted world states that will be traversed if this deterministic trajectory is followed. The system then embarks upon this plan, updating its belief state and checking, at each step, that the most likely world state (according to the belief distribution) is equal to the predicted state. If it is not, the cycle begins again by planning from the current most likely state.

With the MLS assumption and the assumption of deterministic actions, belief replanning is significantly simpler than solving a POMDP, and has been successfully applied in realistically-sized robot navigation domains (Nourbakhsh, Powers, & Birchfield 1995; Cassandra, Kaelbling, & Kurien 1996). Belief replanning has also been developed independently in the model-based diagnosis world and embodied in the Livingstone (Williams & Nayak 1996) and Burton systems (Williams & Nayak 1997). In these cases, belief update is performed via a model-based diagnosis system. The resulting diagnosis and replanning

system is intended to exhibit sub-second response times to determine the state of a complex system such as a spacecraft and determine the set of actions required to reconfigure it. Livingstone was successfully demonstrated in the Remote Agent (Bernard *et al.* 1998) and in several other applications (Kurien, Nayak, & Williams 1998). Livingstone avoids solving an unrestricted STRIPS planning problem by assuming that the plan required to reach a goal state consists of a single set of actions performed in parallel. Burton allows plans that require action sequences. To achieve its performance, Burton assumes it's possible to avoid negative interactions between mode reassignments. That is to say, given a set of modes that need to be reassigned, one can perform the actions to reassign a mode variable without undoing any reassignments that have been done, and without making any remaining reassignments impossible.

Since belief replanning uses a simplified deterministic action model, it will not take into account less likely outcomes of an action that result in negative reward or disaster as the MDP-based solution will. Given the strong bias of our models to the expected outcome of actions, this would appear to be a reasonable approach to avoid solving the underlying MDP. Unfortunately, belief replanning does not take into account the effect of actions in the case where the most likely state is not the true state.

## 11.2 Conformant Planning

Conformant planning is a generalization of deterministic planning wherein the task is to generate a plan that moves a system from any one of a number of possible initial states to a state that satisfies a set of goal predicates. In addition, actions may have uncertain outcomes and sensing actions are not available. The computational challenge of conformant planning lies in the fact that the effects of a plan when executed in one state may be different and highly undesirable when the plan is executed in a different state. Thus one cannot choose an action based on its desired effect given one possible initial state of the system (called a *world* in the conformant planning literature) without in some way considering its unintended effects when it is executed in all other possible initial states.

One traditional approach to conformant planning has been to consider the effects of each action under consideration across all worlds simultaneously. The CGP planner (Smith

& Weld 1998) creates a Graphplan-style planning graph (Blum & Furst 1995) for each world and adds mutual exclusion constraints between them. When an action is selected for inclusion in the plan, its effects across all worlds are simultaneously captured by the multiple planning graphs. CMBP (Cimatti & Roveri 1999) encodes the possible initial states of the world into a binary decision diagram (BDD). An action in a plan maps a BDD that represents a set of worlds onto a new BDD that represents the outcome of the action on each world in the initial BDD. All actions are applied to the initial world BDD and all resulting BDD's until a BDD containing only goal states is found. The path of actions leading to the goal BDD is the conformant plan. GPT (Bonet & Geffner 2001) also considers how an action maps a set of possible states onto a set of resulting states, but relies upon search heuristics rather than compact state set encodings to achieve efficiency. It uses A\* search in the space of world sets rather than breadth-first search as used by CMBP. An admissible heuristic for the A\* search is formed using a fully observable version of the planning problem. Intuitively, the cost of reaching a goal state from a set of states is approximated by the maximum cost of reaching the goal state from any state in the set.

Techniques with a generate and test flavor have also been attempted. In C-PLAN (Castellini, Giunchiglia, & Tacchella 2001) a possible plan is sequence of actions that reaches the goal from at least one initial world. A valid plan is a sequence of actions that achieves the goal from each initial world, and for which every action's preconditions are met in each world. The valid conformant plans are thus a subset of the possible plans. Intuitively, C-PLAN encodes the planning domain and goal as a propositional formula and allows a satisfiability procedure to choose an initial world and possible plan for that world. The plan is then tested for validity as a conformant plan. The main effort of this approach is in limiting the number of possible but invalid plans the planner generates. An additional generate-and-test style conformant planner is presented in (Guere & Alami 1999).

### **11.3 Decision Theoretic Planners**

In general, classical planners have not considered problems involving safety constraints or considered selecting a subset of the goal criteria to satisfy. However, a number of decision theoretic planners have been developed that attempt to maximize the utility of the plans they

generate. Boutilier, Dean and Hanks provide an excellent overview of the field (Boutilier, Dean, & Hanks 1995). Williamson and Hanks (Williamson & Hanks 1996; 1994a; 1994b) have investigated notions of selecting plan flaws in an order that increases plan utility. Applying such a planner to the SCOPE formulation would require a metric on the value of each goal, initial state and safety constraint and a method for combining values across these categories. MAXPLAN finds the plan with the maximum likelihood of achieving a goal (Majercik & Littman 1998). Like SCOPE, MAXPLAN compiles its planning domain to a boolean constraint representation, but uses a probability rather than disjunction to represent uncertainty.

#### 11.4 MDP-based Heuristics

A POMDP with completely deterministic observations (*i.e.*, each observation reveals the true state of the apparatus) is a Markov decision process, or MDP. Given complete observability, the exact solution to an MDP is a policy that specifies the optimal action to take given that the outcome of an action is non-deterministic, but will be known once the action is taken. Such a policy can be obtained quickly even for models with hundreds of thousands of states.

In MDP-based POMDP algorithms, a suboptimal solution to the POMDP is developed from the optimal solution to the corresponding MDP via the MLS, or *most likely state* assumption. Given the belief state, this heuristic finds the world state with the highest probability. It then uses the MDP policy to select and execute the action that would be optimal if the current state and the state resulting from the action were directly observable. This is of course a heuristic, since the state that is most likely given only the current observations may not be the true state, and the action that is optimal given complete knowledge of the state (*e.g.*, whether or not a fuel valve is leaking) may not be optimal given only a distribution over the possible states.

The strength of the MDP heuristic is that it takes into account the uncertainty in the results of actions, thus penalizing actions that might take the system into or near an undesirable state. This strength is undermined by the infinitesimal interpretation of the transition system: the nominal transitions dominate to the extent that non-deterministic outcomes of the action are only considered when conditioning on observations forces it. In addition, as



failures are modeled as occurring from any state, it's not clear how one would act in order to avoid precipitating failure. Thus the main strength of the MDP heuristic, the ability to take into account possible unexpected outcomes of an action before there is an evidence to suggest they may have occurred, seems wasted in this framework. Perhaps more importantly, this heuristic requires solution of the underlying MDP. While an MDP with tens of thousands of states can be solved, the state space of the models we seek to operate on precludes explicit solution of the corresponding MDP.

## Chapter 12

# Conclusions

In this concluding chapter, we review the contributions of the thesis, identify several areas for future work, and end with a few concluding remarks.

### 12.1 Contributions

Chapter 3 through Chapter 6 of the thesis develop a novel framework for diagnosis of complex systems over time. The principal contributions of this portion of the thesis include:

- Development of a novel representation for diagnosis of complex systems over time. This representation allows the belief state at any point in the history of the system being diagnosed to be incrementally generated in most-likely-first order.
- Development of approximations of the diagnostic representation that exhibit low growth or no growth over time.
- Development of a new conflict-based diagnosis algorithm, *CoverTrack*, which is optimized for finding all failures of the same likelihood.
- Implementation of these ideas in the form of *L2*, a diagnosis framework that allows use of *CoverTrack* or more traditional conflict-directed best-first search to perform diagnosis.
- Demonstration of *L2* a range of realistic data generated from NASA control problems.

Chapter 7 and Chapter 8 of the thesis develops a new approach to conformant planning. The principal contributions of this portion of the thesis include:

- Development of fragment-based planning, a novel, incremental approach to the conformant planning problem.
- Development of anytime conformant planning, where a plan that satisfies a single initial world of the conformant planning problem is available immediately, and the number of worlds satisfied by the plan continues to grow as computation time allows.
- Implementation of this system in the form of fragPlan, and demonstration of its operation on a NASA domain and conformant planning problems from the literature.
- An analysis of fragPlan relative to a set of conformant planners available at the time of publication of this thesis, demonstrating fragPlan is computationally competitive.
- Demonstration that, to our knowledge, fragPlan is the fastest conformant planner capable of generating plans with parallel actions

Chapter 9 and Chapter 10 of the thesis address the problem of which plan to generate when no plan to meet all requirements of a planning problem can be found in a timely fashion. The principal contributions of this portion of the thesis include:

- Characterization of the problem of finding a good plan given an unsatisfiable planning problem as constraint suspension and the search for a pareto optimal constraint set.
- Introduction of a family of search strategies for controlling this search, including anytime strategies and strategies that attempt to learn the structure of each problem instance.
- Implementation of these ideas in SCOPE, the Safe, Conformant, Planning Engine.
- Demonstration and analysis of a subset of these search strategies on a variety of planning domains.

## 12.2 Future Work

In some sense, this thesis is only an initial exploration of the ideas in *CoverTrack*, fragPlan and SCOPE. We believe a significant improvement in their capabilities and performance can be add with a moderate amount of additional work.

- First, we believe a significant improvement to the performance of fragPlan may be had

by implementing fragPlan on top of a partial order planner instead of Blackbox. Blackbox assigns actions to specific time steps in the planning horizon. If an action is not constrained to occupy a specific time step by the problem, Blackbox must assign one arbitrarily. This assignment may be inconsistent with the needs of fragments generated for other worlds. For example, suppose when fragPlan generates its initial fragment, Blackbox assigns action A to time step 0. Now suppose in the second fragment, we must insert an action before A. In this case, fragPlan will need to backtrack or restart in order to generate an initial fragment where A occupies a time step other than 0. Our experience is that this type of plan failure accounts for a significant number of the fragments generated and time used in many fragPlan runs. A partial order planner does not assign actions to time steps. Rather, it maintains a set of links that specify the order in which the actions of a plan must be executed. Thus it is always possible to insert an action before or after another, so long as it is consistent with the pre- and post-conditions of the actions involved.

- We have only scratched the surface of the space of possible search control strategies for SCOPE. The addPlan, dropPlan and binPlan strategies add or remove constraints to the planning problem based upon a fixed schedule. A number of adaptive variations can easily be implemented in the existing SCOPE framework. For example, we believe a worthwhile variation of addPlan would consider how much time its current planning attempt took to succeed when determining how many constraints to add to the planning problem. Thus when the current constraint set is quite easy, additional constraints are added rapidly. Similarly, a worthwhile variation of dropPlan is to determine how many goals or worlds were satisfied when the current planning attempt failed. If the most recent call to fragPlan satisfied very few worlds, for example, then constraints should be dropped rapidly. In addition, we have implemented but not yet explored the strategy of tracking which constraints are causing plan failure during each planning attempt. Suppose we are given a planning problem where some worlds or goals are unsatisfiable due to failure. Tracking this information will allow dropPlan variations of SCOPE to determine which worlds or goals are unsatisfiable and drop them from the planning problem.

- The *CoverTrack* algorithm can be significantly improved with a minor adjustment to its implementation. Recall that this algorithm generates all consistent diagnoses of the same likelihood. To do so, it first generates the set of possibly consistent diagnoses by finding a covering for the set of conflicts that have been discovered during diagnosis thus far. Each of these possibly consistent diagnoses differs from each conflict, thus does not contain a known conflict. Each possibly consistent diagnosis is then checked for consistency, which potentially reveals additional conflicts. Recall that when multiple failures relating to the same components occur over time, the computation time for *CoverTrack* rises exponentially with the number of failures. Intuitively, *CoverTrack* is generating many variations of the same possibly consistent diagnosis that all share an inconsistent set of assignments. This occurs only because we generate all diagnoses that cover the current conflict set before testing if any of them are consistent. In Chapter 6, *CoverTrack* generates 2964 possibly consistent diagnoses for the ISSP example of Figure 6.14, and all but 8 are found to be inconsistent. Suppose rather than generating 2964 possibly consistent diagnoses from an initial conflict, we generate only half of the possibly consistent diagnoses, 1482, and check them for consistency. The inconsistent diagnoses that are eliminated will reveal a number of conflicts. If we then attempt to cover the conflicts again, these additional conflicts will allow us to avoid generating many of the other 1482 diagnoses we might have generated. This suggests a very simple strategy of incrementally generating part of the covering set for the initial set of conflicts, then testing them for consistency to augment the conflict set and further constrain the conflict covering problem.

### 12.3 Living With Failure

In recent years, we have enjoyed the efficiency benefits of increased automation of information-centric processes, from online business-to-business auctions, to timely and customized provision of private and government services, to being able to get directions to or a review of almost any business establishment or cultural event from any location. To bring the same kind of fluidity, timeliness and adaptability to our interactions with the

physical world has the potential to make the physical world safer, wealthier and more responsive to the individuals that inhabit it. Unfortunately, physical systems will always fail, especially those made to be affordable or created out of commodity components. Thus any dream of making the physical world as responsive to our needs and desires as our on-line world of information must include an ability for physical systems to detect, diagnose and plan around failures in their operation or their interactions with the environment.

## References

- Bajwa, A., and Sweet, A. 2002. The livingstone model of a main propulsion system. Technical Report TR03.04, Research Institute for Advanced Computer Science (RIACS), Mountain View, CA.
- Bernard, D. E.; Dorais, G. A.; Fry, C.; Jr., E. B. G.; Kanefsky, B.; Kurien, J.; Millar, W.; Muscettola, N.; Nayak, P. P.; Pell, B.; Rajan, K.; Rouquette, N.; Smith, B.; and Williams, B. C. 1998. Design of the remote agent experiment for spacecraft autonomy. In *Proceedings of IEEE Aerospace*.
- Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of the Seventeenth International Joint Conference On Artificial Intelligence (IJCAI-01)*.
- Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference On Artificial Intelligence (IJCAI-95)*, 1636–1642.
- Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-2000)*.
- Bonet, B., and Geffner, H. 2001. Gpt: A tool for planning with uncertainty and partial information. In *Workshop on Planning with Uncertainty and Partial Information, International Joint Conference On Artificial Intelligence (IJCAI-2001)*.
- Boutilier, C.; Dean, T.; and Hanks, S. 1995. Planning under uncertainty: Structural assumptions and computational leverage. In *EWSP*.
- Boyen, X., and Koller, D. 1998. Tractable inference for complex stochastic processes. In *Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, 33–42.
- Bylander, T. 1991. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference On Artificial Intelligence (IJCAI-91)*, 274–279.

- Cassandra, A.; Kaelbling, L. P.; and Kurien, J. 1996. Discrete bayesian uncertainty models for mobile-robot navigation. In *IROS96*.
- Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2001. Improvements to sat-based conformant planning. In *ECP2001*.
- Cheng, H.-T. 1988. *Algorithms for Partially Observable Markov Processes*. Ph.D. Dissertation, University of British Columbia.
- Cimatti, A., and Roveri, M. 1999. Conformant planning via model checking. In *ECP*, 21–34.
- Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. In *Journal of Artificial Intelligence*, volume 13, 303–338.
- D. Putnam, and R. Davis. 1960. A computing procedure for quantification theory. *Journal of the ACM* 7(3):201–215.
- de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130. Reprinted in (Hamscher, Console, & de Kleer 1992).
- de Kleer, J., and Williams, B. C. 1989. Diagnosis with behavioral modes. In *Proceedings of the Eleventh International Joint Conference On Artificial Intelligence (IJCAI-89)*, 1324–1330. Reprinted in (Hamscher, Console, & de Kleer 1992).
- Dearden, R., and Clancy, D. 2002. Particle filters for real-time fault detection in planetary rovers. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, 1–6.
- Doucet, A. 1998. On sequential simulation-based methods for bayesian filtering. Technical Report CUED/F-INFENG/TR. 310, Cambridge University Department of Engineering.
- Dressler, O., and Struss, P. 1992. Back to defaults: Characterizing and computing diagnoses as coherent assumption sets. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*.
- Friedman, N., and Halpern, J. Y. 1999. Modeling belief in dynamic systems part ii: Revision and update. *Journal of Artificial Intelligence* 10.



- Giunchiglia, E. 2000. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *KR'2000*.
- Goldszmidt, M., and Pearl, J. 1992. Rank-based systems: A simple approach to belief revision, belief update, and reasoning about evidence and actions. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, 661–672.
- Gomes, C. P.; Selman, B.; McAloon, K.; and Tretkoff, C. 1998. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 208–213.
- Guere, E., and Alami, R. 1999. A possibilistic planner that deals with non-determinism and contingency. In *Proceedings of the Sixteenth International Joint Conference On Artificial Intelligence (IJCAI-99)*.
- Hamscher, W.; Console, L.; and de Kleer, J. 1992. *Readings in Model-Based Diagnosis*. San Mateo, CA: Morgan Kaufmann.
- Haslum, P., and Jonsson, P. 1999. Some results on the complexity of planning with incomplete information. In *Proceedings of the 5th European Conference on Planning (ECP-99)*, volume 1809, 308–318. Springer Verlag.
- Hauskrecht, M. 2000. Value-function approximations for partially observable markov decision processes. *Journal of Artificial Intelligence Research* 13:33–94.
- Isard, M., and Blake, A. 1998. Condensation – conditional density propagation for visual tracking. *International Journal of Computer Vision* 29(1):5–28.
- Joslin, D. E., and Clements, D. P. 1999. Squeaky wheel optimization. *Journal of Artificial Intelligence* 10.
- Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101.
- Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic,

- and stochastic search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-96)*.
- Kautz, H., and Selman, B. 1999. Unifying sat-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference On Artificial Intelligence (IJCAI-99)*.
- Kurien, J., and Nayak, P. P. 2000. Back to the future with consistency based trajectory tracking. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*.
- Kurien, J.; Nayak, P. P.; and Williams, B. 1998. Model-based autonomy for robust mars operations. In *Founding Convention of the Mars Society*.
- Kushmeric, N.; Hanks, S.; and Weld, D. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1-2):239–286.
- Langley, P. 1992. Systematic and nonsystematic search strategies. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems (AIPS-92)*, 145–152.
- Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference On Artificial Intelligence (IJCAI-97)*, 366–371.
- Littman, M.; Cassandra, A.; and Kaelbling, L. 1995. Learning policies for partially observable environments: Scaling up. In *Machine Learning: Proceedings of the Twelfth International Conference*, 362–370.
- Majercik, S. M., and Littman, M. L. 1998. MAXPLAN: A new approach to probabilistic planning. In *Artificial Intelligence Planning Systems*, 86–93.
- McDermott, D. 1987. A critique of pure reason. *Computational Intelligence* 3:151–160.
- Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*.
- Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103:5–47.

- Nayak, P. P., and Williams, B. C. 1997. Fast context switching in real-time propositional reasoning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-97)*.
- Nourbakhsh, I.; Powers, R.; and Birchfield, S. 1995. Dervish: An office-navigating robot. *AI Magazine Summer*.
- Pell, B.; Gamble, E. B.; Gat, E.; Keesing, R.; Kurien, J.; Millar, B.; Nayak, P. P.; Plaunt, C.; and Williams, B. C. 1998. A hybrid procedural/deductive executive for autonomous spacecraft. In *Proceedings of the Second International Conference on Autonomous Agents*.
- Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, 189–197.
- Peot, M. 1998. *Decision-Theoretic Planning*. Ph.D. Dissertation, Stanford University.
- Probst, C. . 1993. *Ford Fuel Injection & Electronic Engine Control : All Ford/Lincoln-Mercury Cars and Light Trucks 1988 to 1993*. Bentley Publishers.
- Schwabacher, M.; Samuels, J.; and Brownston, L. 2002. The nasa integrated vehicle health management technology experiment for x-37. In *Proceedings of SPIE AeroSense*. SPIE.
- Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26.
- Smith, D. E., and Weld, D. S. 1998. Conformant graphplan. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-98)*.
- Sondik, E. J. 1971. *The Optimal Control of Partially Observable Markov Processes*. Ph.D. Dissertation, Stanford University.
- Struss, P. 1997. Fundamentals of model-based diagnosis of dynamic systems. In *Proceedings of the Fifteenth International Joint Conference On Artificial Intelligence (IJCAI-97)*, 480–485.
- Thrun, S.; Langford, L.; and Verma, V. 2002. Risk sensitive particle filters. In *Proceedings of the Neural Information Processing Systems Conference*.

- Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-96)*, 971–978.
- Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Proceedings of the Fifteenth International Joint Conference On Artificial Intelligence (IJCAI-97)*.
- Williamson, M., and Hanks, S. 1994a. Optimal planning with a goal-directed utility model. In *International Conference on Artificial Intelligence Planning Systems*, 176–181.
- Williamson, M., and Hanks, S. 1994b. Utility-directed planning. In *American Association for Artificial Intelligence*.
- Williamson, M., and Hanks, S. 1996. Flaw selection strategies for value-directed planning. In *International Conference on Artificial Intelligence Planning Systems*, 237–244.
- Zhang, N. L.; Lee, S. S.; and Zhang, W. 1999. A method for speeding up value iteration in partially observable markov decision processes. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI-99)*, 696–703.