

LEARNING WHAT TO INSTRUCT: ACQUIRING KNOWLEDGE FROM
DEMONSTRATIONS AND FOCUSSED EXPERIMENTATION

by

Richard Harrington Angros, Jr.

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

May 2000

Copyright 2000

Richard Harrington Angros, Jr.

Acknowledgements

Getting a PhD is a large undertaking and requires the help and support of many people.

I would like to thank my advisor Lewis Johnson. Obviously, he provided a great deal of advice and guidance. He was careful to offer guidance in such a way that I had the freedom to make my own decisions. Having gone through this process, I am now much better able to make these decisions.

I would like to thank Jeff Rickel with whom I worked on the VET project. He was always available to offer comments and suggestions.

I would like to thank my committee members for graciously providing time and effort. My committee consisted of Lewis Johnson, Jeff Rickel, Martin Frank, Allen Munro, Paul Rosenbloom and Skip Rizzo. They each provided insightful comments and suggestions. I also want to thank them for their patience and for reading two drafts of this document.

I would like to thank Jeff Rickel for the STEVE tutor. He was always willing to answer questions about it and to fix problems. I would also like to thank everyone else who worked on STEVE, especially Ben Moore and Marcus Thiébaux.

I would like to thank the people at BTL for the VIVIDS authoring tool. I would like to particularly thank Allen Munro and Quentin Pizzini. They not only answered questions, but they also made changes to VIVIDS in order to support my work. These changes included fixing software problems that only I experienced as well as the ability for an external program to save and restore the state of a VIVIDS model.

I would also like to thank the people at Lockheed Martin who worked on the VET project and on the VISTA Viewer, which managed the display of the simulated environment. Particularly, I would like to thank Randy Stiles and Laurie McCarthy.

A number of people helped me design the empirical evaluation. These people include Lewis Johnson, Jeff Rickel, Martin Frank, Paul Rosenbloom, Allen Munro, Skip Rizzo and Gallen Buckwalter. I would especially like to thank Skip Rizzo and Gallen Buckwalter because they work in a different part of the university and are not associated with the VET project.

When I first started attending USC, Seymour Ginsburg kindly gave of his time and offered me some very good advice.

I would like to thank the Information Sciences Institute's (ISI) Soar group. The Soar group was a place to explore new ideas and to hear a variety of perspectives. Some current and former members include Jonathan Gratch, Randy Hill, Gal Kaminka, Jihie Kim, Lewis Johnson, Jeff Rickel, Paul Rosenbloom, Ian Stobie, Ben Smith, Bongham Cho and Karl Schwamb.

I would like to thank additional people here at ISI. Lorna Zorman helped me when I first started. I also want to thank Kate LaBore and Erin Shaw. Additionally, I want to thank fellow grad students and officemates Ali Erdem and Chon Yi.

I would like to thank Hughes Aircraft and Raytheon for supporting my efforts to get this degree. I wish to thank Hughes for a fellowship, and I want to thank some of my managers at Hughes: Natacha Estrada, Mike Stemig, Dave Manz and Diana Chu.

I would also like to thank the Office of Naval Research for funding the VET project with grant N00014-95-C0179 and AASERT grant N00014-97-1-0598.

Finally, I would like to thank my family and parents. They've provided a great deal of support and encouragement.

Contents

Acknowledgements	ii
List Of Tables	xi
List Of Figures	xii
Abstract	xv
1 Introduction	1
1.1 Motivation	2
1.1.1 A Model of Teaching	2
1.1.2 What Activities Must a Tutor Perform?	3
1.1.3 The Required Knowledge	4
1.1.4 Acquiring Knowledge from Experts	5
1.1.5 Heterogeneous Tutoring Environments	5
1.1.5.1 Advantages of a Heterogeneous Architecture	5
1.1.5.2 Limitations of a Heterogeneous Architecture	6
1.2 The Problem	6
1.3 Addressing the Knowledge Acquisition Bottleneck	7
1.4 The Basic Approach	8
1.5 Related Work	9
1.6 Contributions	10
1.7 Organization of the Thesis	11
2 Using Diligent	12
2.1 Properties of a Simple Procedure	12
2.2 What Does the Author Need to Know?	13
2.2.1 Concepts Needed for Basic Use	14
2.2.2 Concepts Needed for Advanced Use	15
2.3 Using Diligent to Author a Procedure	18
2.3.1 The Procedure to be Authored	18
2.3.2 Authoring the Procedure	19
2.3.2.1 Creating the Procedure	19
2.3.2.2 Specifying the Initial State	19
2.3.2.3 Demonstrating the Procedure	20

2.3.2.4	Creating a Procedure from the Demonstration	23
2.3.2.5	The Initial Procedure	24
2.3.2.6	Refining the Procedure with Experiments	25
2.3.2.7	Additional Authoring Activities	28
2.4	Summary	28
3	Diligent – Its Problem and Approach	30
3.1	The Problem	30
3.1.1	Requirements	30
3.1.1.1	Reducing the Instructor’s Effort	31
3.1.2	Constraints on the Domain Knowledge	31
3.1.3	Interface with the Environment	33
3.2	Input and Output	37
3.2.1	Input	37
3.2.1.1	Definitions of Basic Data Types	38
3.2.2	Output	40
3.2.2.1	Procedures	41
3.2.2.2	Operators	43
3.3	How Diligent Works	46
3.3.1	Processing Demonstrations	46
3.3.2	Heuristics	48
3.4	Where to Look for More Information	50
4	Processing Demonstrations	51
4.1	The Authoring Process	51
4.2	Types of Demonstrations	52
4.3	Data Structures	53
4.3.1	Prefixes	53
4.3.2	Demonstrations	54
4.3.3	Paths	55
4.3.4	Steps	55
4.3.5	Revisiting the Representation of Procedures	57
4.4	Assumptions about How the Instructor Demonstrates	57
4.5	About this Chapter’s Extended Example	59
4.6	Authoring a New Procedure	59
4.6.1	Creating a New Procedure	59
4.6.2	Setting Up the Initial State	60
4.6.3	Demonstrating the Procedure	60
4.6.4	Creating Primitive Steps	60
4.6.5	Converting the Demonstration into a Path	64
4.6.6	A Second Demonstration	66
4.6.6.1	Setting Up the Demonstration’s Initial State	67
4.6.6.2	Performing The Demonstration	67
4.6.6.3	Processing the Demonstration	68
4.6.7	Generating a Plan	69
4.6.7.1	Guessing the Procedure’s Goals	70

4.6.7.2	Deriving Step Relationships	71
4.7	Creating a Hierarchical Procedure	80
4.7.1	Internally Simulating A Subprocedure	82
4.7.2	Continuing the Running Example	86
4.7.3	A Nested Procedure Definition	87
4.7.4	Sensing Actions	88
4.7.5	Demonstrating the Nested Procedure	90
4.8	The Completed Procedure	91
4.8.1	Information Provided by the Instructor	94
4.8.1.1	Generating default descriptions	95
4.9	Complexity	95
4.10	Related Work	97
4.10.1	Natural Language Versus Direct Manipulation	97
4.10.2	Programming By Demonstration	98
4.10.2.1	Procedure Representation	98
4.10.2.2	Basic Techniques	99
4.11	Summary	100
5	Learning Operators	102
5.1	Additional Requirements	103
5.2	Heuristics	104
5.3	About this Chapter's Examples	105
5.4	Data Structures	105
5.4.1	Preconditions as a Version Space	106
5.5	Creating a New Operator	109
5.6	Positive and Negative Examples	112
5.7	Refining Preconditions	113
5.7.1	Refining Preconditions with Positive Examples	114
5.7.2	Refining Preconditions with Negative Examples	116
5.7.2.1	Discriminating Between Effects	120
5.8	Putting it all Together	123
5.8.1	Determining How to Process Effects	124
5.8.2	Adding a New Effect	126
5.8.3	Splitting an Effect in Two	129
5.9	Complexity Analysis	132
5.9.1	Scalability	135
5.10	Related Work	136
5.11	Summary	137
6	Experimenting	139
6.1	The Problem	140
6.1.1	Requirements	140
6.2	Background	141
6.2.1	Focused versus Unfocused	142
6.2.2	Supervised versus Unsupervised	142
6.2.3	Experimenting with Plans	143

6.3	Input	144
6.4	Diligent's Approach	145
6.5	The Procedure Being Used	146
6.6	The Algorithm	147
6.6.1	What Was Learned From the Experiment	151
6.7	Complexity Analysis	153
6.7.1	Scalability	155
6.8	Related Work	156
6.8.1	The Self-Explanation Effect	156
6.8.2	Other Systems	156
6.9	Summary	157
7	Empirical Evaluation	159
7.1	Hypotheses	159
7.2	The Three Versions of Diligent	160
7.3	Usability Analysis	162
7.4	Experimental Method	164
7.4.1	Independent Variable	164
7.4.2	Test Subjects	164
7.4.3	Dependent Variables	166
7.4.3.1	Measuring Errors in Plans	167
7.4.4	Test Procedure	168
7.4.5	The Procedures Being Authored	171
7.4.6	Data Analysis	172
7.5	Results	173
7.5.1	Results of Background Questionnaire	173
7.5.2	Time Spent Training	175
7.5.3	Logical Edits	176
7.5.4	Errors	179
7.5.4.1	Errors in Identifying Steps	179
7.5.4.2	Errors of Omission	180
7.5.4.3	Errors of Commission	183
7.5.4.4	Total Errors	185
7.5.5	Total Required Effort	189
7.5.6	Time Spent Authoring	189
7.5.7	Subjective Impressions	189
7.6	Discussion	193
7.6.1	Assumptions About Test Subjects	193
7.6.2	Discussion of Background Questionnaire	194
7.6.3	Discussion of Training Time	195
7.6.4	Discussion of Logical Edits	196
7.6.5	Discussion of Errors in Identifying Steps	196
7.6.6	Discussion of Errors of Omission	197
7.6.7	Discussion of Errors of Commission	198
7.6.8	Discussion of Total Errors	198

7.6.9	Discussion of Total Required Effort	199
7.6.10	Discussion of Time Spent Authoring	199
7.6.11	Discussion of Subjective Impressions	200
7.7	Reviewing the Claims	201
7.8	Observations	204
7.9	Summary	204
8	Analysis and Future Work	206
8.1	Perspectives for Understanding Demonstrations	206
8.2	Assumptions	208
8.2.1	Easier to Relax	208
8.2.2	Harder to Relax	211
8.3	Limitations	214
8.3.1	Coordinated Simultaneous Actions	214
8.3.2	When Pre-State and Post-State Values are Independent	214
8.3.3	Transitive Dependencies	216
8.4	Extensions	217
8.4.1	Procedural Representation	217
8.4.1.1	Multiple Methods for Performing a Procedure	217
8.4.1.2	Conditional Plans	219
8.4.1.3	Disjunctive Goal Conditions	219
8.4.2	Authoring	220
8.4.2.1	Additional Types of Demonstrations	220
8.4.2.2	Continuous/Parameterized Actions	221
8.4.2.3	Types of Mental Attributes	221
8.4.2.4	Inferred Attributes	222
8.4.3	Learning	222
8.4.3.1	Simple Extensions	222
8.4.3.2	More Involved Extensions	223
8.4.4	Experimentation	224
8.4.4.1	Simple Extensions	224
8.4.4.2	More Involved Extensions	225
8.5	Summary	225
9	Related Work	226
9.1	The Presentation of Examples	226
9.1.1	Felicity Conditions	226
9.1.2	Presenting a Sequence of Examples	229
9.2	Intelligent Tutoring Systems	232
9.2.1	Computer Aided Instruction	232
9.2.2	Who is the Author	233
9.2.3	Approach to Authoring	234
9.2.4	Easier Data Entry	235
9.3	Learning From Demonstrations	237
9.3.1	Programming By Demonstration	237
9.3.2	Detailed Domain Models	238

9.3.3	Procedure Recognition	239
9.3.4	University of Michigan Soar Group	239
9.3.5	Approach to Experimentation	240
9.3.6	Systems that Learn Operators	241
9.3.7	Other Work	242
10	Conclusion	244
10.1	Summary of the Approach	244
10.2	Contributions	245
10.3	Evaluation	246
10.4	Future Work	246
	Reference List	248
	Appendix A	
Implementation	261
A.1	Architecture	261
A.2	Maintenance of Agenda	262
A.3	Providing Feedback About Diligent's Beliefs	264
	Appendix B	
Evaluation Materials	266
B.1	Background Questionnaire	266
B.2	Procedure Representation Description	268
B.3	The Procedure Representation Worksheet	274
B.4	Worksheet Answers	276
B.5	The Post-Test	277
B.6	The Directions Given Subjects	279
B.7	The List of Attribute Values	283
B.8	Labeled Pictures of the HPAC	287
B.9	Procedure Descriptions	292
B.9.1	High Condensate Level Shutdown	292
B.9.2	Overload Relay Tripped	293
B.10	Desired Procedures	294
B.10.1	High Condensate Level Shutdown	294
B.10.2	Overload Relay Tripped	297
B.11	Practice Procedure	299
B.12	Practice Procedure Solution	300
	Appendix C	
Evaluation Data	302
C.1	Background Questionnaire	303
C.2	Impressions of Diligent	304
C.2.1	Experimental Condition EC_1	305
C.2.2	Experimental Condition EC_2	306
C.2.3	Experimental Condition EC_3	308

C.3	Authoring	309
C.3.1	Experimental Condition EC_1	311
C.3.2	Experimental Condition EC_2	314
C.3.3	Experimental Condition EC_3	317
C.4	Session Log	320

Appendix D

	How to Use Diligent	329
D.1	Starting to Specify a Procedure	330
D.2	Demonstrations	331
D.2.1	Chapter Goals	331
D.2.2	Setting the Initial Environment State	331
D.2.3	Adding Steps	332
D.2.4	Operator Descriptions	335
D.2.5	Add More Steps	335
D.2.6	End Demonstration	335
D.2.7	Additional Demonstrations	337
D.2.8	Choosing a Previous Step	338
D.3	Adding Steps to a Procedure	339
D.3.1	Chapter Goals	339
D.3.2	Adding Steps	339
D.3.3	Choosing a Previous Step	340
D.3.4	Selecting an Action	340
D.3.5	Operator Descriptions	342
D.3.6	Selecting Operator Effects	343
D.3.7	Adding Operator Effects	345
D.3.8	Selecting Operator Effect's Revisited	350
D.3.9	Add a Couple More Steps	350
D.4	Editing a Procedure	351
D.4.1	Chapter Goals	351
D.4.2	Review: Reaching the Procedure Modification Menu	351
D.4.3	Procedure Graphs	352
D.4.4	Looking at a step	354
D.4.5	Operator Effect Menu	356
D.4.6	Precondition Window	357
D.4.7	State Change Window	357
D.4.8	Modifying Preconditions	358
D.4.8.1	Using the Operator Effect menu	358
D.4.8.2	Using the Step Prerequisites menu	358
D.4.9	Updated Procedure Graph	358
D.4.10	Updated Step Modification Menu	361
D.4.11	Dependencies Menu	363
D.4.12	Looking at the Causal Link Menu	364

List Of Tables

3.1	Where Topics are Covered	50
6.1	Changes to proc1 's Preconditions	152
7.1	Distribution of Subjects Based on Sex and Language	165
7.2	Activities Performed By Subjects	169
7.3	Background ANOVA Tests	174
7.4	Background Means and Standard Deviations	174
7.5	Background ANOVA Tests	175
7.6	Training Time Means and Standard Deviations	176
7.7	Linear Regression on Total Training Time	177
7.8	Logical Edit Analysis	177
7.9	Means and Standard Deviations on Invalid Steps	179
7.10	Errors of Omission Analysis	180
7.11	Errors of Commission Analysis	183
7.12	Total Error Analysis	185
7.13	Total Required Effort Analysis	187
7.14	Analysis of Time Spent Authoring	190
7.15	Subjective Impressions	192
7.16	Summary of Results	203
A.1	Status Values Used by Diligent	265
C.1	EC_1 Impressions about Authoring	305
C.2	EC_1 Impressions about Demonstrations and Experiments	305
C.3	EC_2 Impressions about Authoring	306
C.4	EC_2 Impressions about Demonstrations and Experiments	306
C.5	EC_3 Impressions about Authoring	308
C.6	EC_1 Procedure 1 Authoring Information	311
C.7	EC_1 Procedure 2 Authoring Information	312
C.8	EC_1 Time Spent on Activities	313
C.9	EC_2 Procedure 1 Authoring Information	314
C.10	EC_2 Procedure 2 Authoring Information	315
C.11	EC_2 Time Spent on Activities	316
C.12	EC_3 Procedure 1 Authoring Information	317
C.13	EC_3 Procedure 2 Authoring Information	318
C.14	EC_3 Time Spent on Activities	319

List Of Figures

2.1	Procedure Description Menu	19
2.2	Resetting the Environment's State	19
2.3	The Front of the HPAC	21
2.4	Operator Description Menu	23
2.5	Hypothesized Goal Conditions	24
2.6	Hypothesized Goal Condition Details	25
2.7	The Initial Dependencies	26
2.8	The Dependencies After Experimentation	27
3.1	Input/Output	36
3.2	Syntax of Basic Data Types	38
3.3	An Action-Example	40
3.4	Example Plan procA	42
3.5	Example Operator toggle-motor	44
3.6	Processing a Demonstration	47
4.1	First Demonstration's Action-Examples	61
4.2	Creating a Primitive Step	62
4.3	First Demonstration	64
4.4	Initializing a Path	65
4.5	The Initial Path	65
4.6	Using a Prefix	67
4.7	The Second Demonstration's Prefix	68
4.8	The Second Demonstration	68
4.9	Adding a Demonstration to a Path	69
4.10	Updated Path	69
4.11	Deriving Goals from a Path	72
4.12	Goal Conditions Derived from Path	72
4.13	Computing Step Relationships	73
4.14	The Operators	74
4.15	Identifying a Path's Effects	76
4.16	Skeleton of Procedure	77
4.17	Computation of Causal Links	78
4.18	Causal Links	79
4.19	Computation of Additional Ordering Constraints	81
4.20	Ordering Constraints	82

4.21	The Plan for Procedure proc1	83
4.22	Simulating a Subprocedure	84
4.23	Results from Simulating Step proc1-6	87
4.24	The Subprocedure's Prefix	88
4.25	Computing State Changes Caused by Earlier Steps	89
4.26	Subprocedure Demonstration	92
4.27	The Plan for Subprocedure proc2	93
4.28	The Top Level Procedure	94
5.1	Preconditions for Starting a Car	104
5.2	Relationship between the Precondition Concepts	106
5.3	An Operator	107
5.4	Algorithm for Creating New Operator	110
5.5	Input for Creating New Operator	111
5.6	A New Operator	111
5.7	Some Positive and Negative Examples	112
5.8	Refining Preconditions with a Positive Example	114
5.9	Using a Positive Example	115
5.10	Potentially Needed Conditions	116
5.11	Refining Preconditions with Negative Example	117
5.12	Using Negative Examples	118
5.13	Discriminating Between Effects	121
5.14	An Example of Discriminating Between Effects	122
5.15	Refining an Operator with an Example	125
5.16	An Example for Assigning Delta-State Conditions to Effects	126
5.17	Creating a New Effect	127
5.18	An Example of Creating a New Effect	129
5.19	Splitting an Effect	130
5.20	An Example of Creating a New Effect	131
6.1	A Hierarchical Procedure	147
6.2	The Top Level Experimentation Algorithm	147
6.3	Generating Skip-Step Experiments	148
6.4	Performing Experiments	149
6.5	The Stack of Actions to Perform	150
7.1	Graphs of Logical Edits	178
7.2	Graphs of Errors of Omission	181
7.3	Graphs of Errors of Commission	184
7.4	Graphs of Total Errors	186
7.5	Graphs of Total Required Effort	188
7.6	Graphs of Time Spent Authoring	191
8.1	An Attribute whose Post-State is Independent of its Pre-State	215
8.2	Incompatible Paths	218

A.1	The VET Software Architecture	261
A.2	The STEVE Tutoring Agent	263
B.1	Procedure with Steps in Specification Order	269
B.2	Procedure with Steps Ordered by Dependencies	270
B.3	Operator with Two Effects	271
B.4	Example Steps	272
B.5	Procedure Example2's Dependencies	273
D.1	Main Learning Menu	330
D.2	Main Learning Menu "Editing" Options	330
D.3	Procedure Description Menu	330
D.4	Simulation Configuration Menu	331
D.5	Communications Bus Monitor Window	332
D.6	Additional Environment Changes	332
D.7	Demonstration Menu	333
D.8	Environment before Demonstration	333
D.9	Environment after Demonstration	334
D.10	Operator Description Window	335
D.11	Soar Processing an Action	336
D.12	Demonstration Version of Procedure Modification Menu	337
D.13	Demonstration Type Menu	338
D.14	Previous Step Menu	339
D.15	Manual Editor Version of Procedure Modification Menu	340
D.16	Previous Step Menu	341
D.17	Action Selection Menu	341
D.18	Operator Description Window	342
D.19	Effect Selection Menu Before Effects Defined	342
D.20	Initial Operator Effect Menu	344
D.21	Precondition Attribute List	345
D.22	Attribute Value Input Window	345
D.23	Precondition Value Window	346
D.24	Updated Operator Effect Menu	347
D.25	Updated Effect Selection Menu	349
D.26	Main Learning Menu	351
D.27	Procedure Graph from "Ordering relationships"	352
D.28	Procedure Graph showing "execution order"	353
D.29	Step Modification Menu	354
D.30	Operator Effect Menu	356
D.31	Precondition Window	357
D.32	State Change Window	357
D.33	Step Prerequisites Menu	359
D.34	Incorrect Procedure Graph	360
D.35	Step Modification Menu with Error	361
D.36	Dependencies Menu	363
D.37	Causal Link Menu	364

Abstract

One way that people can learn procedural tasks (e.g. machine maintenance) is by performing them in a simulation of the domain. Simulation-based training not only allows repetition, but also allows exposure to situations that would be dangerous or expensive in the real-world.

However, for simulation-based training system to teach students more effectively, the training system needs to know the procedures that students are learning. Unfortunately, it has been difficult to acquire this type of knowledge.

This dissertation looks at using machine learning techniques to acquire procedures that can be used for tutoring human students. The work focuses on exploiting access to a simulation of the domain. Because we assume that an automated tutoring program will already know general principles for teaching, we focus on learning “what” to teach rather than “how.” The approach learns general-purpose operators and outputs knowledge of what to teach in the form of hierarchical partially ordered plans.

The approach is implemented by a system called *Diligent*. Procedures are specified by having human authors demonstrate them. To demonstrate, a human performs the procedure by directly manipulating a graphical representation of the simulation. This technique could be used by domain experts, who may not be programmers or expert knowledge engineers. However, there is a problem: a single demonstration may be insufficient for understanding the causal relationships between a procedure’s steps, while requiring many demonstrations would take too much time and effort.

Diligent attempts to overcome this problem by using a novel approach for understanding demonstrations. Although *Diligent* may start with no domain knowledge, it can reduce the requisite number of demonstrations by performing autonomous experiments that exploit the domain knowledge embedded in the simulation. These experiments identify the causal dependencies between a demonstration’s steps by removing a step from the demonstration and observing how this affects later steps. These experiments are augmented by heuristics that focus on the dependencies between steps.

We performed an evaluation on human subjects that tested the benefits of our approach. As dependent variables, we measured the amount of work performed and the number of errors. The results suggest that using demonstrations and experiments together is better than using just demonstrations. The results also suggest that demonstrating a procedure is better than using an editor to declaratively specify it. The benefits of both demonstrations and experiments appeared to be greater on more complicated procedures.

Chapter 1

Introduction

A common activity that people engage in is performing procedures. Procedures include such things as programming a VCR or following a cooking recipe. Knowledge of how to perform procedures is called *procedural knowledge*. Besides the ability to follow fixed recipes, procedural knowledge also includes knowing how to adapt procedures to a given situation. For example, when repairing an engine, a part might break or an unrelated problem might be discovered.

For procedures that involve manipulation of the real-world (e.g. machine maintenance), there are advantages to having human students learn procedures by performing them in a simulation of the domain. Not only can the students gain exposure to many training problems, but they can also experience a variety of unusual situations. Training with simulations is very useful when real-world training episodes are expensive or dangerous (e.g. machine maintenance or surgery).

However, not all types of training are equally useful. One effective method of training is tutoring. When compared to conventional classroom instruction, studies have shown that tutoring students one-on-one can improve their achievement by two standard deviations [Blo84, Ana83, Bur83]. Unfortunately, human tutors are expensive, and their availability is usually very limited.

The limited supply of human tutors can be overcome by using a computer program as an automated tutor (e.g. STEVE [RJ99]). An automated tutor can free a human instructor for more specialized instruction by assuming many of his normal duties. In performing the instructor's normal duties, an automated tutor can use general-purpose knowledge about teaching. However, general-purpose knowledge alone is insufficient because an automated tutor also needs domain specific knowledge about the procedures being taught.

Unfortunately, it has not been easy to acquire this type of knowledge. In fact, the general problem of acquiring domain knowledge from experts has been called the *knowledge acquisition bottleneck* [Hof87].

Fortunately, in the case of simulation-based training, considerable domain knowledge may already be contained in a simulation. By exploiting access to the domain knowledge embedded in the simulation, the acquisition of procedures can be made easier. This dissertation looks at how a software program (or *agent*) can use machine learning techniques and access to a simulation in order to learn the procedural knowledge necessary for an automated tutor to teach human students.

This chapter is roughly organized as follows. First, we will motivate our research; this involves discussing what we mean by teaching, what knowledge is needed, and the difficulties in acquiring this knowledge. Second, we will discuss an approach for learning procedures. We will then finish by discussing contributions and providing a high level overview of related work.

1.1 Motivation

1.1.1 A Model of Teaching

In this thesis, we assume that an automated tutor will use an apprenticeship style of teaching [CBN89]. Apprenticeship is a traditional method for learning procedural knowledge and involves “learning-through-guided-experience” [CBN89]. During an apprenticeship, an apprentice (or student) learns by observing, copying and interacting with a master (or tutor). Learning a procedure typically has two phases:

1. A student observes a tutor demonstrate a procedure. Observing the demonstration allows the student to create an initial conceptual model of the procedure.
2. The tutor then monitors the student as the student performs the procedure. While the student is performing the procedure, the tutor can provide help and reminders. At this stage, a potential problem is that the student may only be able to repeat the procedure by rote. To make sure that the student knows how to adapt the procedure to different situations, he may perform it several times from different initial states.

As a student gains proficiency in a domain, the procedures that the student performs tend to become more complex. Moreover, because the student is more capable, the tutor provides less help and direction.

1.1.2 What Activities Must a Tutor Perform?

To teach a procedure, an automated tutor must be able to perform a number of activities. The ability to use knowledge for multiple purposes is a characteristic of Intelligent Tutoring Systems (ITS) [Wen87]. The activities that we are interested in will be illustrated with the following example.

When an air compressor is running, the condensation of water can cause pressure to build up internally. When the pressure is too high, the compressor shuts down in order to avoid damage. To restart the compressor, a human operator does the following. He resets the compressor and opens a valve that allows the water to drain. He then restarts the motor. After the pressure is relieved, he shuts the valve.

A tutor needs to be able to demonstrate procedures to students. A demonstration involves performing a sequence of actions that will achieve the procedure's goals. Consider the above example. The tutor could demonstrate restarting compressor by performing the following sequence of actions: open the valve, press the reset button, toggle the start button, and close the valve.

A tutor also needs to be able to monitor students as they perform a procedure. A human student might legitimately perform the actions in a different order than any demonstration. Thus, the tutor needs to know more than the sequences of actions used in demonstrations; the tutor also needs the knowledge to recognize valid sequences of actions. For example, when the tutor demonstrated restarting the compressor, the tutor opened the valve before pressing the reset button. However, the student could have instead pressed the button before opening the valve. If the tutor didn't recognize that the two actions were independent, the tutor might believe that the student has to press the button again after opening the valve.

As a tutor is demonstrating a procedure or monitoring a student, the tutor needs to be able to answer the student's questions. To avoid confusing the student, the tutor should not provide incomplete or incorrect answers. Questions that students might ask include ones about how to perform a procedure and why an action is needed [Dav84]. Questions of this type include the following.

- As a student performs the procedure, he might ask which actions are currently applicable. For example, when starting the restart procedure, a student might ask what actions can he do now. The tutor could then tell the student that he could press the reset button or open the valve.

- A student might also ask why an action is being performed. For example, a student might ask why he needs to toggle the start button. The tutor might give two reasons: it turns on the motor, and it results in a normal internal pressure.
- A student could also ask why the state changes produced by an action are important. For example, if a student asked why a normal internal pressure is important, the tutor might say that the pressure needs to be normal before closing the valve and that having normal pressure is one of the goals of the procedure.

The tutor also needs some capability to respond to student errors, to unusual situations, and to unexpected state changes. An example of an unusual situation is when the procedure is started with the valve already open. An example of a student error is when a student opens the valve but shuts it before starting the motor. In this case, the student would have to reopen the valve before he could start the motor. An example of an unexpected state change is when a student opens the valve but then spends a few minutes taking care of more urgent business. In the mean time, someone could walk by, notice the valve is open, and shut it. The tutor should be able to react to the valve being unexpectedly shut.

1.1.3 The Required Knowledge

In this dissertation, the knowledge of how to perform procedures will be represented using hierarchical partially ordered plans [RN95]. Not only is this representation commonly used by the Artificial Intelligence (AI) community, but the representation has been often used in work that focuses on describing procedures to humans. This research on describing procedures has focused on topics such as providing concise descriptions [You97], selecting rhetorical relations to express the relationships between actions [VM95, Van93], multi-lingual instruction generation [DHP⁺94, PV96, PVF⁺95], and representing relations that hold between pairs of actions [Pol90, Di 94, Bal93].

This procedural representation has several properties that are needed to provide adequate explanations [ME89, You97].

- Large, complicated procedures can be decomposed into a sequence of smaller, simpler and logically coherent subprocedures. For example, consider a procedure that teaches someone to drive to the store. One subprocedure might involve starting a car, and another subprocedure might involve stopping at a stop light.

- The representation describes causal dependencies between steps that can be used to answer questions about how to perform a procedure or why is an action needed. Recall that these are the types of questions that we discussed in Section 1.1.2.

1.1.4 Acquiring Knowledge from Experts

Before a procedure can be used, it needs to be acquired from a domain expert. Unfortunately, domain experts often have great difficulty encoding knowledge in a form that a computer program can use [Mus93, Gai87, EEMT87, Hof87]. The process tends to be tedious, time consuming and error prone. One problem is that programs often require a very structured and formal representation. This representation may be very different than how the expert thinks about the domain. Additionally, the techniques used for specifying a procedure may be very different than how the instructor would teach a human student. For these reasons, the process of specifying procedures may seem awkward and unnatural.

1.1.5 Heterogeneous Tutoring Environments

Instead of addressing the general knowledge acquisition problem, this thesis addresses knowledge acquisition in heterogeneous, simulation-based tutoring environments. By heterogeneous, we mean that the system contains multiple software components (e.g. automated tutor and simulation) and that these components may have been created by different people and organizations.

This problem is easier than the general knowledge acquisition problem because of access to a simulation, which embodies knowledge of the domain. Although a simulation provides a executable model of the domain, simulations do not know the procedures that we are trying to teach.

1.1.5.1 Advantages of a Heterogeneous Architecture

This type of modular, heterogeneous architecture has advantages. Consider the situation when the interfaces between components are stable. Old components can be replaced by newer ones without modifying the other components. Because components can be developed separately, it may be possible to spend more time on each, especially if it appears that a component can be reused.

The modularity of components should make transferring the tutoring environment to a new domain easier. For example, the simulation for the old domain might be replaced with a simulation for the new domain. Because simulations are often written during product

design and evaluation, an existing simulation for the new domain might be available. The system's modularity should also allow easier transfer of general-purpose components (e.g. automated tutors).

One reason for transferring an automated tutor to another domain is that the tutor may contain a lot of reusable, general knowledge about teaching. However, without knowledge of a domain's procedures, a tutor could not teach.

1.1.5.2 Limitations of a Heterogeneous Architecture

An important characteristic of heterogeneous systems is that individual components may contain specialized knowledge and have little access to the internal knowledge of other components. Different components are likely to have been created by different people and organizations, and some of these people may have little or no contact with each other. Moreover, different components may use different representations and programming languages.

The limited access to another component's knowledge can impose restrictions on the types of operations that another component (e.g. an authoring tool) can perform on a simulation. Ideally, another component (e.g. an authoring tool) could extract knowledge from the simulation by putting the simulation in a particular state, performing the type action that a human would (e.g. press a button), and observing the result. While a simulation should be able to support the type of actions that humans perform, the simulation may not allow another component to independently set the values of individual attributes. The rationale for this limitation is that other components may not know which states of the simulation are valid. Arbitrary changes to the simulation's state could result in inconsistent attribute values. Even worse, arbitrary state changes could put the simulation in an unsupported, invalid state and result in invalid behavior. This limitation is more important when a simulation models something (e.g. a machine) with many constraints on valid attribute values.

1.2 The Problem

This thesis deals with acquiring procedures from domain experts in a heterogeneous, simulation-based tutoring environment. To do this, we will want to meet a few general requirements.

- We want to reduce the effort required from the domain expert. This includes using techniques that will reduce the number of number of questions that the user is asked.

- We want to use methods that could be used by a large class of users. In particular, we are interested in someone who teaches human students (i.e. an *instructor*). Although an instructor is a domain expert, he may not have expertise in programming or knowledge engineering.
- We want an approach that can be easily transferred to a new domain. This means that the approach should require little explicit encoding of domain knowledge.

1.3 Addressing the Knowledge Acquisition Bottleneck

To address these requirements, we will make some assumptions about the components of the heterogeneous tutoring system. In particular, we assume that an instructor has access to a simulation of the “real-world” domain. In this document, we will use the term *environment* when referring to the components of the tutoring system that simulate the domain. We assume that the environment provides two important capabilities: it provides a graphical representation of the domain, and it allows the software program that learns procedures to interact with the simulation that controls the environment.

This thesis focuses on how a software program (or *agent*) can use machine learning techniques and access to the environment in order to reduce the effort required from an instructor. The approach addresses the knowledge acquisition bottleneck in two ways.

- The instructor can communicate with the agent at a high level. The communication is at a high level because the graphical representation of the domain allows the instructor to demonstrate procedures in manner similar to how he would demonstrate them to a human student.
- The agent’s ability to observe and interact with the simulation allows the agent to use machine learning techniques for extracting information from the simulation. The ability to observe demonstrations allows the agent to observe how a demonstration changes the simulation’s state. Furthermore, the ability to interact with the simulation allows the agent to improve its understanding by performing experiments.

One limitation of this approach is that the agent may not have enough data to learn an entirely correct procedure. Instead, the approach provides an heuristic aid to the instructor by learning procedures that are “reasonably” correct.¹ Given such a procedure, an instructor can then use his domain knowledge to refine it.

¹A procedure is “reasonably” correct in the sense that it should be close to the correct procedure.

Because observing a demonstration may not provide enough knowledge to learn a correct procedure, it is desirable for an agent to acquire more knowledge. One approach is to ask the instructor questions that clarify areas of uncertainty. However, instead of asking the instructor questions, the agent could have attempted to answer its own questions by performing experiments. This thesis leaves questioning the instructor as an area for future work and instead focuses on understanding demonstrations by performing experiments.

1.4 The Basic Approach

The basic approach for learning procedures involves integrating demonstrations and experiments. The agent initially learns a procedure by observing a demonstration. The agent then refines the procedure by performing experiments that are derived from the demonstration.

The instructor demonstrates procedures by using a mouse to directly manipulate a graphical representation of the domain. When demonstrating, the instructor performs a sequence of actions (e.g. pressing a button) that manipulate the domain. These actions are referred to as the demonstration's *steps*, and this approach is called Programming By Demonstration [C⁺93]. It should be easy for an instructor to provide demonstrations because he performs the procedure just like a human student would.

While the instructor is demonstrating, the agent is constantly learning. However, the agent may not have enough data to learn the correct procedure. In particular, the agent may not have enough data to correctly identify how earlier steps in a demonstration establish preconditions of later steps. To compensate for its lack of data, the agent identifies likely preconditions. Even though likely preconditions may be incorrect, they aid the instructor by identifying a small set of items on which he should focus.

To provide more data for learning, the agent then attempts to improve its understanding of a demonstration by experimenting. Experiments provide data by transforming one demonstration into multiple similar demonstrations. The agent experiments by replaying a demonstration while skipping a step. This allows the agent to observe how eliminating the step affects later steps. The observations are used to identify missing or incorrect preconditions. By extracting knowledge from the simulation, experiments reduce the amount of data that the instructor needs to provide.

Because there may be relatively little data, the instructor may need to interact further with the agent. This interaction may include looking at the agent's knowledge, testing the procedure or providing additional demonstrations.

As an artifact of learning procedures, the approach learns operators. An *operator* models an action that the instructor performs during a demonstration. To do this, an operator identifies the preconditions necessary for the action to produce a given state change. Operators are used to contain the reusable domain knowledge associated with a demonstration's steps. Unlike steps, which are belong to one procedure, operators can be reused in many procedures.

To help understand how operators differ from steps, consider a procedure that uses a pump to drain a tank of water. One of the steps needed to drain the tank is pressing the start button, which starts the motor. While pressing the button is a step in this specific procedure, knowledge of how pressing the button affects the pump could possibly reused in other procedures. Diligent stores this reusable knowledge in operators.

This approach for authoring procedures has been implemented in a system called *Diligent* [AJR97]. The system is called Diligent because of its tenacity in attempting to understand demonstrations.

1.5 Related Work

Diligent differs from earlier work because it solves a different problem: it is designed to exploit the presence of a simulation so that it can learn the knowledge necessary to provide good explanations while requiring minimal data from the user.

Some systems learn to perform procedures by only learning how to react to the current state (e.g. Instructo-Soar [HL95], IMPROV [Pea96] and Metamouse+ [MWM94]). While it is possible for systems that learn this type of knowledge to provide good explanations, this type of system is unlikely to be able to explain how a procedure's steps depend on each other. One reason for this limitation is that a system might not be able to directly examine some of its knowledge (e.g. production memory in Soar [LNR87]). Another reason is that the knowledge may not be structured in a way that supports good explanations [Cla86].

Some machine learning systems require a lot of data. This may mean that the learning algorithm requires a lot of data (e.g. MSDD [OC96]), but it may also mean that system does not make the most effective use of the data that it is given. For example, OBSERVER [Wan96c], like Diligent, learns operators; but OBSERVER doesn't consider why a demonstration's steps are sequenced in a given order.

Unlike Diligent, some systems do not need access to a simulation (e.g. Disciple [TH96], KidSim [SCS94], MARVIN [SB86], and ALVIN [KW88]). These systems may have a very effective interaction with users, but they do not extract knowledge from a simulation

with autonomous experiments. If these systems were used on Diligent’s problem, users would have to answer questions that provide the same knowledge that Diligent learns from experiments.

Other approaches require the ability to make arbitrary changes to the state of a simulation while ignoring the constraints on valid attribute values (e.g. PET [PK86]). In machine maintenance domains, this type of change to a simulation’s state is either not allowed or is likely to put the simulation in an invalid state.

Another problem is that systems can require an initial domain theory, which may be difficult to acquire (e.g. EXPO [Gil92], ASK [Gru89], CELIA [Red92], LEAP [MMS90], ARMS [Seg87], LEX [MUB83]). Although a domain theory can make learning easier and allow systems to have additional capabilities, someone must spend the requisite amount of time necessary to encode and validate the domain theory.

Finally, some systems use a representation for preconditions that is inappropriate for our problem. The representation may be complicated or difficult to understand (e.g. IMPROV [PL96]). Even if the representation is simpler, the knowledge may still be represented in an unnecessary complex manner (e.g. LIVE [She93]). The complexity of the representation is important because an instructor is less likely to accept a system if he is uncomfortable with how it represents the domain knowledge.

1.6 Contributions

This work focuses on using machine learning techniques to integrate demonstrations and experiments. This integration not only reduces the amount of data that the user has to provide but also makes it easier to provide the data. In this way, the approach supports more efficient authoring of procedures for intelligent tutoring systems. Because the approach exploits access to a simulation, the approach can also be used with little explicit encoding of domain knowledge.

The major contributions are as follows.

- A method that balances the strengths and weaknesses of demonstrations and experiments. Experiments are used to identify missing or unnecessary preconditions, but can more easily identify unnecessary preconditions. For this reason, operators are created during demonstrations using heuristics that have a bias towards creating unnecessary preconditions. While creating operators, the system uses a novel heuristic that focuses on how earlier steps in a demonstration establish preconditions for later

steps. Because experiments compensate for the bias towards creating unnecessary preconditions, Diligent can learn a great deal from a single demonstration.

- A method for performing useful and focused experiments while requiring only minimal knowledge. The approach only needs to know the sequence of steps in a demonstration. The approach exploits the simulation to focus on how the state changes of early steps in a demonstration affect later steps. This approach effectively transforms one demonstration into multiple related demonstrations.

1.7 Organization of the Thesis

The rest of this dissertation is organized as follows.

Chapter 2 presents a high-level description of how to use Diligent. It also discusses the basic concepts that an author would need to know.

Chapter 3 describes the problem more formally and provides an overview of how Diligent works. The problem statement includes high level requirements and Diligent's inputs and outputs. The discussion of how Diligent works describes the main heuristics and the major data flows.

Chapter 4 discusses interaction with instructors and how demonstrations are transformed into plans. This includes the assumptions that Diligent makes about demonstrations. The algorithms in this chapter provide support for learning operators and performing experiments.

Chapter 5 discusses how Diligent learns operators.

Chapter 6 discusses how Diligent generates additional training data by performing autonomous experiments.

Chapter 7 provides an empirical evaluation of Diligent being used by humans.

Chapter 8 brings together the material discussed in the earlier chapters. It discusses how demonstrations, experiments and machine learning are integrated. It also talks about Diligent's assumptions and how easily they could be relaxed. It finishes by discussing limitations and potential extensions.

Chapter 9 discusses related work.

Chapter 10 provides a short summary.

Appendix A describes implementation details.

Appendix B contains materials used for evaluating Diligent.

Appendix C contains the data found during the evaluation.

Appendix D contains tutorial material that describes how to use Diligent.

Chapter 2

Using Diligent

Diligent is a tool that supports authoring of procedures that perform tasks such as machine maintenance. The steps in these tasks include actions such as reading gauges, pushing buttons and turning handles. While some of these procedures might seem simple and intuitive, there is a need for training because there is a lot of room for human error. Obviously, automating this type of training requires someone to specify procedures with a software tool.

One obstacle to authoring procedures is that all tools require knowledge of some concepts. One advantage of Diligent is that it only requires an author to know relatively few concepts.

This chapter illustrates how to use Diligent and describes the concepts that an author would need to know.

2.1 Properties of a Simple Procedure

In order to illustrate the basic properties of procedures, we will consider a simple procedure for starting a car. To start a car, a person needs to do the following.

1. Open the door.
2. Get in.
3. Shut the door.
4. Put on the seat belt.
5. Put the key in the ignition.
6. Start the car by turning the key.

However, it is not that simple – sometimes the car will not start. Perhaps, the battery is bad, or perhaps, the transmission is in drive rather than park.

How could people learn this type of task? One approach is simple trial and error, but this is not likely to be effective with an automobile. In fact, it could be dangerous. Another approach is to repeat a demonstration of the task by rote. However, this approach is fragile and inflexible. Consider what happens in a slightly different situation. What if one of the steps is not needed? For example, if the key is already in the ignition, do you need to take the key out and put it back in? What if one of the steps is counter-productive? For example, suppose that you always turn on the radio when you start the car. If the radio is already on, pressing the radio's on/off button will turn it off. A better approach for learning a task is to build a simple model; this model could then be used to identify the purpose of each step and the relationships between steps.

Consider a simple model of starting a car.

The abstract goal is to start the car in a manner that allows you to drive safely. To achieve this abstract goal, you need to satisfy a number of conditions: the door needs to be shut, you need to be wearing the seat belt, and the motor needs to be running.

There are also goals for each step. For example, you turn the key in order to start the car. If the motor is already running, you don't need to turn the key.

Individual steps may also have preconditions. For example, turning the key will not start the motor if the transmission is in drive or the battery is bad.

There are also dependencies between the steps because some steps may influence later steps. For example, you need to put the key in the ignition before turning the key. However, other steps can be performed in any order. For example, it doesn't matter whether you first shut the door or put on the seat belt.

2.2 What Does the Author Need to Know?

By authoring a procedure with Diligent, an author creates a model similar to one described in the previous section.

However, to do this, the author (or instructor) needs to understand certain concepts. As in most software systems, knowledge of more concepts is needed by an expert user than by a minimally competent user.

2.2.1 Concepts Needed for Basic Use

A minimally competent user needs to know about the types of knowledge needed to represent a procedure.¹

- The state of the world is represented by conditions. A condition indicates that a given attribute has a specific value. For example, the position of the door is shut, or the seat belt is fastened.
- A procedure has an abstract description, which is used to describe the procedure to students. For example, the previous section's procedure allowed you to "start the car in manner that allows you to drive safely."
- The purpose of a procedure is to achieve a set of goal conditions. This means that some of the environment's attributes need to have certain values. For example, the goal conditions of the car starting procedure are "the door is shut, the seat belt is fastened, and the motor is running."
- Performing a procedure involves performing a sequence of actions that are called steps. For example, starting the car involves "opening the door, getting in, . . ."
- Each step has an English description that is used to describe the step to students. For example, the act of opening a door could be described as "opening the door."
- Steps have preconditions, which are conditions that need to be true immediately before the step is performed. For example, before turning the key, the transmission should be in park.
- There are dependencies between steps because some steps cause changes that are preconditions of later steps. These dependencies are called causal links. A causal link identifies an attribute value that is a precondition for one step and is established by an earlier step. For example, the act of putting the key in the ignition causes the key's location to be in the ignition, and the presence of the key in the ignition is precondition for turning the key. Thus, there should be a causal link between the step that inserts the key and the step that turns the key.

¹This discussion ignores concepts needed by system components other than Diligent. For example, we will not consider the concepts needed to manipulate the environment's graphical interface.

- Each causal link has an English description that can be given to students. For example, the presence of the key in the ignition might be described as “the key is in the ignition.”
- The concept of causal links can be extended to include the initial state in which the procedure starts and the end of the procedure when all the procedure’s goal conditions are satisfied. A causal link involving the initial state indicates that a precondition of a step is satisfied when the procedure starts. For example, the car starting procedure depends on the transmission being in drive at the start of the procedure. A causal link involving the end of the procedure indicates that a step establishes a goal condition. For example, putting on the seat belt establishes the goal condition that the seat belt is fastened.

A minimally competent user also needs a couple of concepts that involve Diligent’s method of authoring.

- A user demonstrates a procedure by manipulating objects in the environment’s graphical interface.
- Diligent can use demonstrations to generate and perform experiments that are likely to improve the correctness of the preconditions of a procedure’s steps.

2.2.2 Concepts Needed for Advanced Use

To get beyond minimal competence, a user should know additional concepts. Unfortunately, Diligent’s implementation does not always distinguish between basic and advanced concepts. This makes a minimally competent user’s job more difficult because Diligent’s user interface shows advanced concepts that are not needed for basic use.

To get beyond minimal competence, a user should know some additional concepts related to the representation of procedures.

- The actions performed by users are modeled by operators. Operators describe the preconditions necessary for an action to produce some desired state changes. Operators differ from steps in that operators are independent of a given step or procedure. The use of operators allows Diligent to reuse the knowledge contained in an operator by associating the operator with potentially many steps. Each step is associated with one operator, and Diligent uses the step’s operator to identify the step’s preconditions. Suppose the user wants to author procedures for starting a car in different

initial states. For example, there might be a one procedure for when the battery is bad and another procedure for when the transmission is initially in drive. In this case, all procedures for starting the car would share the operator for turning the key in the ignition.

Diligent's user interface expects users to know about operators during demonstrations. To simplify the interaction with the user, Diligent generates a step's name using its operator. If there is no operator for an action, Diligent creates one and asks the user for a name and an English description.

- If an action is performed in different situations, it can cause different state changes. Operators model this by having multiple conditional effects (or *effects*). Each effect identifies preconditions that need to be satisfied for the action to produce a given set of state changes. For example, suppose a radio had an on/off button. When the radio is on, pressing the button turns it off, and when the radio is off, pressing the button turns it on. An operator that models pressing the button should have one effect for turning the radio on and one for turning it off.
- A Step can also have preconditions that are specific to that step. Because these preconditions are independent of the step's operator, the preconditions are associated with the step rather than the operator. For example, a person could read a car's fuel gauge at any time. However, the value shown on the gauge may be invalid when the motor is off. Therefore, if you have a step that reads the fuel gauge, you may want the step to have a precondition that requires the motor to be running.
- When Diligent initially learns a procedure, some preconditions may be incorrect. Diligent can help the user identify problems with preconditions by indicating its confidence in a given precondition. In this way, a user can distinguish between preconditions that are very likely and those that are somewhat likely. (The measures of belief are the s-rep, h-rep and g-rep, which are described in Chapter 3.)
- Sometimes the preconditions of several steps may be satisfied. When this happens, it may not matter which of the steps is performed first, but sometimes it does. This type of dependency between two steps is called an ordering constraint. An ordering constraint indicates which of the two steps to perform first. For example, an ordering constraint might indicate that a car's key needs to be inserted into the ignition before the key is turned. Normally, there is one ordering constraint for every causal link, but sometimes additional ordering constraints are needed. Suppose that

a procedure involved unlocking the glove compartment, removing a map, locking the glove compartment, and then inserting the key in the ignition. It might be possible to take the key out of the lock and put it in the ignition without locking the glove compartment. This problem can be avoided by adding an ordering constraint to prevent the key from being inserted into the ignition before the glove compartment is locked.

Although Diligent's user interface assumes that all users know about ordering constraints, the user interface need not have made this assumption. Users should almost always use the ordering constraints that Diligent automatically identifies.

- Diligent allows users to author a large procedure by hierarchically composing it from smaller procedures. When this happens, the small procedures are treated as steps in the large procedure. When a procedure is used as a step, like any other step, it is considered to have preconditions and produce state changes. Suppose the user wants to create a procedure for driving to a grocery store. In this procedure, the procedure for starting a car might be the first step.

An advanced user should also have some knowledge of how Diligent learns.

- Diligent assumes that changes to the state during a demonstration are likely to be important. Sometimes state changes of early steps in a demonstration are proposed as preconditions of later steps. This approach sometimes causes Diligent to propose unnecessary dependencies between steps. (Diligent can later attempt to correct these unnecessary dependencies by performing experiments.)

An author can use Diligent's bias towards state changes to provide demonstrations that promote more effective learning. The steps in a demonstration should be closely related, and closely related steps should be demonstrated together. If each step in a procedure comes from a different one step demonstration, Diligent does not learn as effectively.

- Diligent's learning can be improved if a given action (e.g. fastening the seat belt) is demonstrated in very different situations. Diligent has Clarification demonstrations that support this type of input (Section 4.2).

2.3 Using Diligent to Author a Procedure

The above discussion describes the concepts that authors should know, but it doesn't describe how an author would use Diligent. This section illustrates how to use Diligent.² To do this, it uses a procedure from the High Pressure Air Compressor (HPAC) domain.³

2.3.1 The Procedure to be Authored

Sometimes high levels of condensation can build up inside the compressor. To avoid damaging the machine, the compressor's condensate drain monitor turns off the motor. To restart the motor, the condensation needs to be drained.

This condensation is built up in one of the separator drain manifold's five valves. There is one valve for each of the compressor's stages. If too much condensation builds up in one of the stages, the alarm light for that stage is illuminated.

For the procedure's initial state, we will assume that the motor has stopped and that alarm lights for the first and second stages are illuminated.

The procedure to restart the compressor has the following steps.

1. Open the second stage valve. This is done by turning a handle that is used to open and shut valves.
2. Move the handle to the first stage valve.
3. Use the handle to open the second stage valve.
4. Reset the compressor by pressing the condensate drain monitor reset button.
5. Start the motor by pressing the motor button on the HPAC's control door panel. This starts the motor and drains the first and second stages. Once the stages are drained, the alarm lights will turn off.
6. Use the handle to close the first stage valve.
7. Move the handle to the second valve.
8. Use the handle to close the second stage valve.

²Appendix D contains a more complete discussion of how to use Diligent.

³Section B.9.1 contains a plan for this procedure.

2.3.2 Authoring the Procedure

Suppose that Diligent initially has no knowledge of the domain.

2.3.2.1 Creating the Procedure

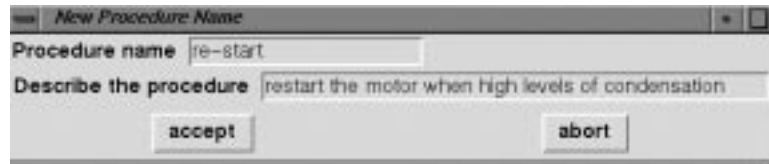


Figure 2.1: Procedure Description Menu

When authoring a procedure, the first thing the instructor does is tell Diligent that he wants to author a procedure. Diligent then brings up the Procedure Description Menu (Figure 2.1). The procedure name is used by the instructor to identify the procedure, while the description contains English text that is presented to students.

In our example, the instructor calls the procedure “re-start” and types in the description “restart the motor when high levels of condensation.” Afterwards, the instructor continues by selecting the “accept” button.

2.3.2.2 Specifying the Initial State

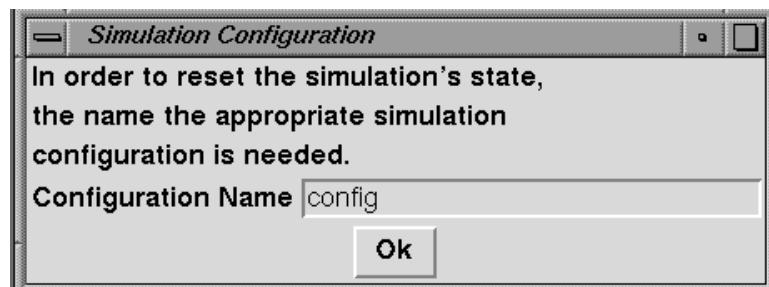


Figure 2.2: Resetting the Environment's State

Before starting the demonstration, the simulation environment needs to be put in an initial state. To do this, the instructor resets the environment to known configuration, which is identified by a text string (Figure 2.2).

Resetting the environment to known state may seem strange, but it is very useful in an educational setting. It supports both teaching and authoring. It allows an automated tutor to reset the environment before demonstrating a procedure to students. It also allows students to start procedures from pre-specified initial states. The ability to reset the state is also useful when providing additional demonstrations and when Diligent experiments with a procedure.

In our example, the instructor enters the configuration “config” and then selects the “Ok” button. This causes Diligent to reset the state of the simulation environment. The instructor will use this new state as the procedure’s initial state.

2.3.2.3 Demonstrating the Procedure

Once the environment is in the desired initial state, the instructor can start demonstrating the procedure. Figure 2.3 shows the environment’s graphical representation of the procedure’s initial state.

To demonstrate, the instructor directly manipulates objects in the graphical interface by selecting them with a mouse. Here is how an instructor could demonstrate the example procedure.

1. The instructor opens the first stage valve by selecting the handle (the cross-shaped object) that is on top of the valve. This causes the object that represents the handle to turn. This new position indicates that the valve is open.⁴
2. The instructor moves the handle to the second stage valve by selecting the second stage valve.
3. The instructor opens the second stage valve by selecting the handle that is on top of the valve.
4. The instructor resets compressor by selecting the condensate drain monitor reset button.
5. The instructor interacts with the simulation to move the view of HPAC to the control door. (In the current implementation, this can be done by pressing a button that is in a different window.) This is not recorded as one of the demonstration’s steps.
6. The instructor starts the motor by selecting the motor button.

⁴One idiosyncrasy of the environment’s graphical interface is that whether a valve is open or shut can only be seen when the handle is on top of the valve.

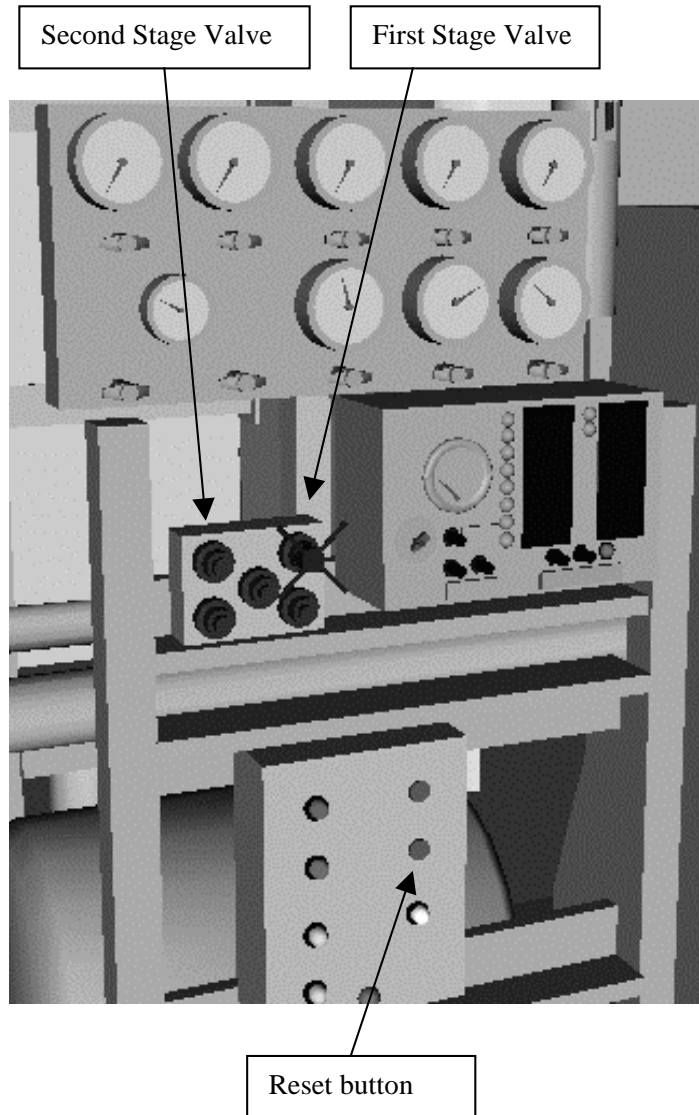


Figure 2.3: The Front of the HPAC

7. The instructor interacts with the simulation to move the view back to the front of the HPAC. (This can also be done by pressing a button that is in a different window.) This is not recorded as one of the demonstration's steps.
8. The instructor closes the first stage valve by selecting the handle that is on top of the valve.
9. The instructor moves the handle to the first stage valve by selecting the first stage valve.
10. The instructor closes the first stage valve by selecting the handle that is on top of the valve.

At this point, the instructor indicates that he is done with the demonstration.

The above discussion of the demonstration does not discuss the creation of operators. Diligent associates each step in the demonstration with an operator, which Diligent uses to model the step's preconditions and state changes. By associating a reusable operator with a step, Diligent is better able to identify the step's preconditions because it can learn from other steps that use the same operator.

Ideally, a minimally competent user would not see or care about operators. For example, a user might focus on the preconditions and state changes of the steps while ignoring how they were identified.

However, Diligent's user interface exposes the concept of operators to instructors. This happens the first time that an instructor demonstrates a given action (e.g. selects the first stage valve). At this point, Diligent asks the instructor to name the operator and to modify or approve Diligent's default description. Once operators have been given names, Diligent uses the name to automatically generate step names. (Note that default names and descriptions could have been generated without consulting the instructor.)

Figure 2.4 shows the naming of the operator for the example procedure's first step. The default description was generated by considering the type of action (i.e. turning the handle) and the object acted upon (i.e. the handle that is used for manipulating valves).

In the example procedure, Diligent learns operators that model the following actions: turning the handle on top of a valve, moving the handle to the second stage valve, pressing the reset button, pressing the motor button, and moving the handle to the first stage valve.



Figure 2.4: Operator Description Menu

2.3.2.4 Creating a Procedure from the Demonstration

So far, we have identified a sequence of steps that make changes to environment's state. However, we have not yet identified what the procedure is supposed to accomplish or how later steps depend on earlier steps.

The first thing the instructor needs to do is tell Diligent to identify some likely goal conditions. A goal condition indicates the value a given attribute must have in order to finish the procedure (e.g. the compressor's motor is running). Because the purpose of a procedure is to change the environment's state, Diligent hypothesizes that the final values of all attributes that change value during the procedure are goal conditions. After Diligent identifies some likely goal conditions, the instructor is shown a menu (Figure 2.5) where he can approve or reject the conditions. In our example, we will assume that the instructor approves all hypothesized goal conditions.

Unfortunately, the goal conditions in Figure 2.5 are a little cryptic because attribute names are being shown. This problem can overcome by selecting a goal condition with the mouse and getting more a more detailed description (Figure 2.6).

Once the goal conditions have been identified, Diligent is able to identify which steps achieve which goal conditions and to determine how later steps are dependent on earlier steps. This calculation could have been done automatically, but for implementation reasons, Diligent requires the instructor to explicitly request this calculation.

Once the goal conditions and the dependencies between steps have been identified, Diligent is able generate a procedure from the demonstration.

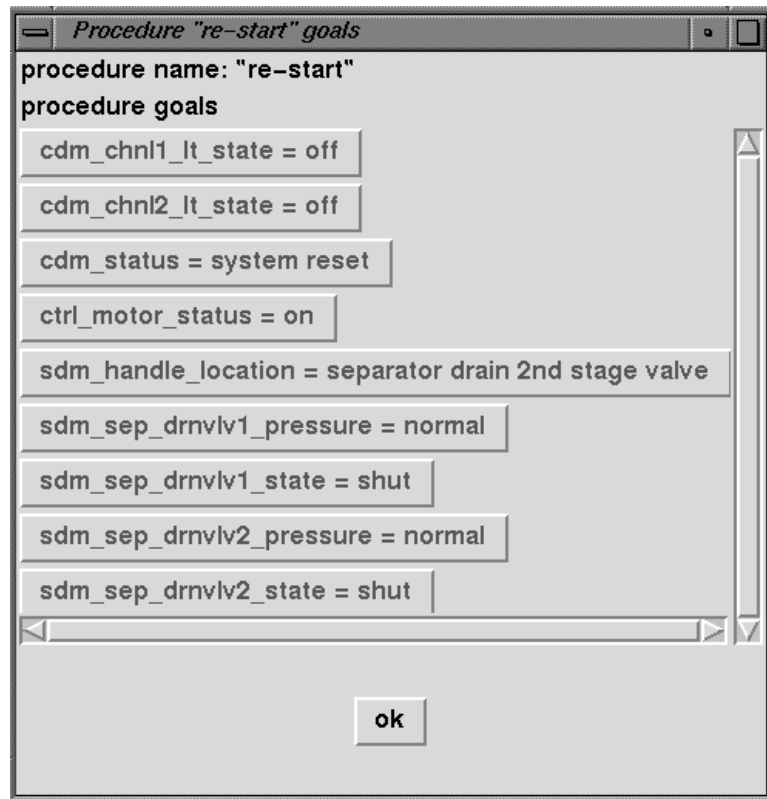


Figure 2.5: Hypothesized Goal Conditions

2.3.2.5 The Initial Procedure

Figure 2.7 shows the dependencies between the procedure's steps when the procedure is first created. The lines represent dependencies between pairs of steps. The steps that the instructor demonstrated are shown as ovals. The ovals contain step names, which are numbered sequentially in the order that the steps were demonstrated. The rectangle labeled **begin-re-start** represents the procedure's initial state. Not shown in the figure is a rectangle labeled **end-re-start** that represents the end of the procedure. A line between a step and the end of the procedure, indicates that the step establishes one of the procedure's goal conditions.

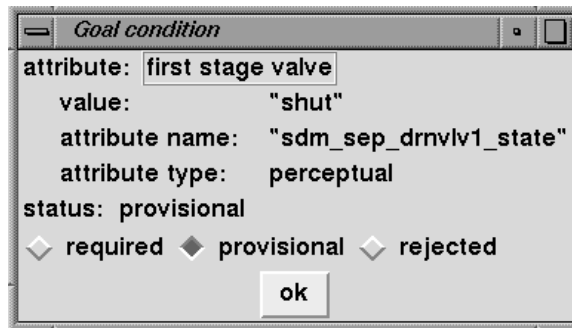


Figure 2.6: Hypothesized Goal Condition Details

To edit the procedure, the instructor can bring up a menu by selecting an oval or a rectangle with the mouse. The details of these menus won't be shown because they emphasize details of the user interface's implementation.⁵

2.3.2.6 Refining the Procedure with Experiments

Because the initial procedure (Figure 2.7) was created with very little data, Diligent had use to heuristics. One heuristic is that state changes caused by earlier steps are likely to be preconditions of some later steps. While the dependencies between steps identified by the heuristics are often valid, the dependencies are not always valid.

Diligent can help alleviate this problem by generating more data for identifying the preconditions of steps. Diligent does this by performing experiments where it performs the procedure while skipping a step. These experiments allow Diligent to observe how the missing step affects later steps.

Some advantages of experiments are that the instructor does not have to perform any additional work and that an experiment's results are gathered from the simulation's executable model of the domain.

In order to keep Diligent's user interface more responsive, Diligent only experiments when told to do so by the instructor. The problem is that experiments require dedicated use of the simulation environment. If Diligent automatically decided to initiate an experiment, then during the experiment, the instructor could not use the environment's graphical interface to perform demonstrations or to test procedures. Furthermore, experiments

⁵Appendix D discusses Diligent's menus.

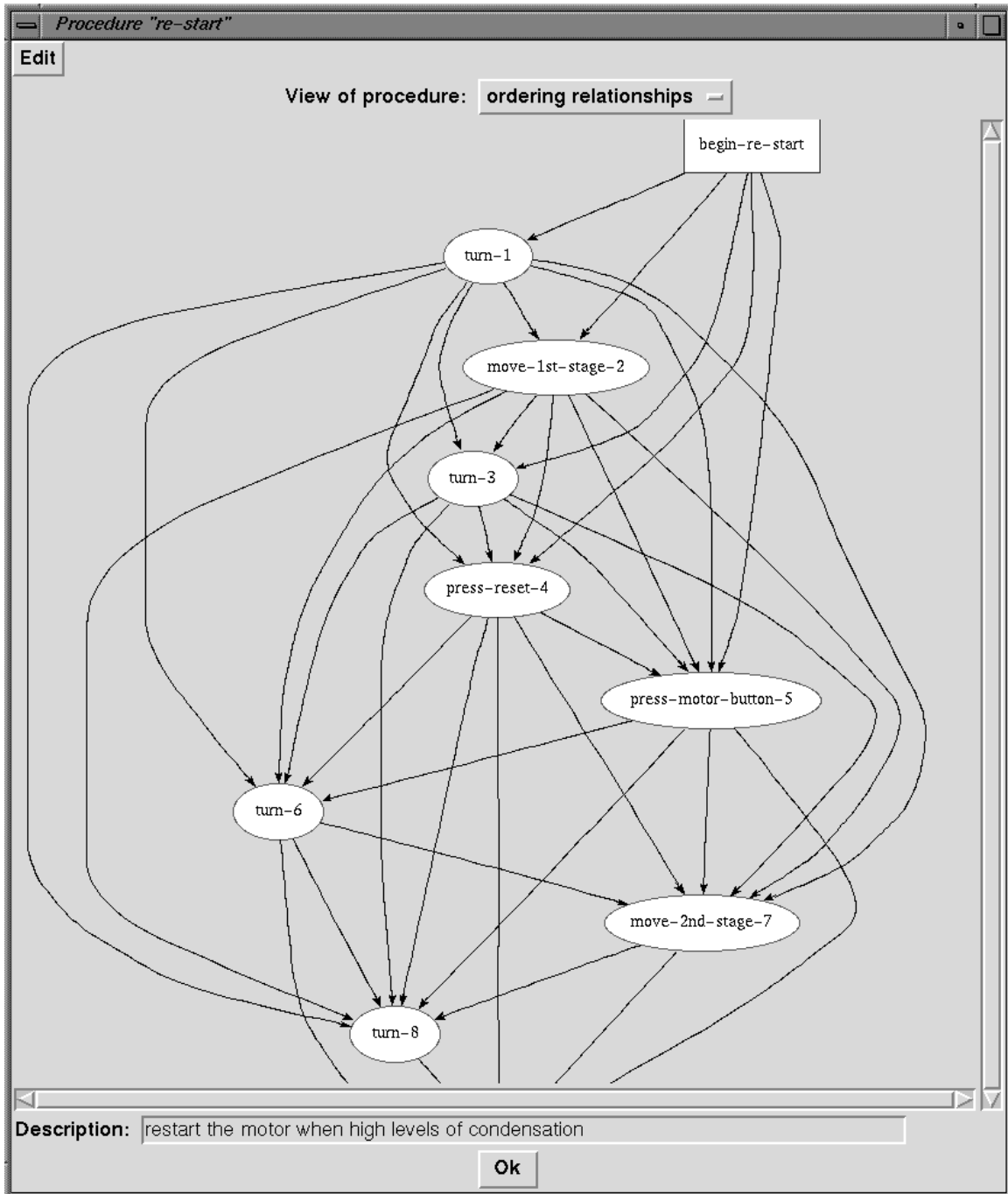


Figure 2.7: The Initial Dependencies

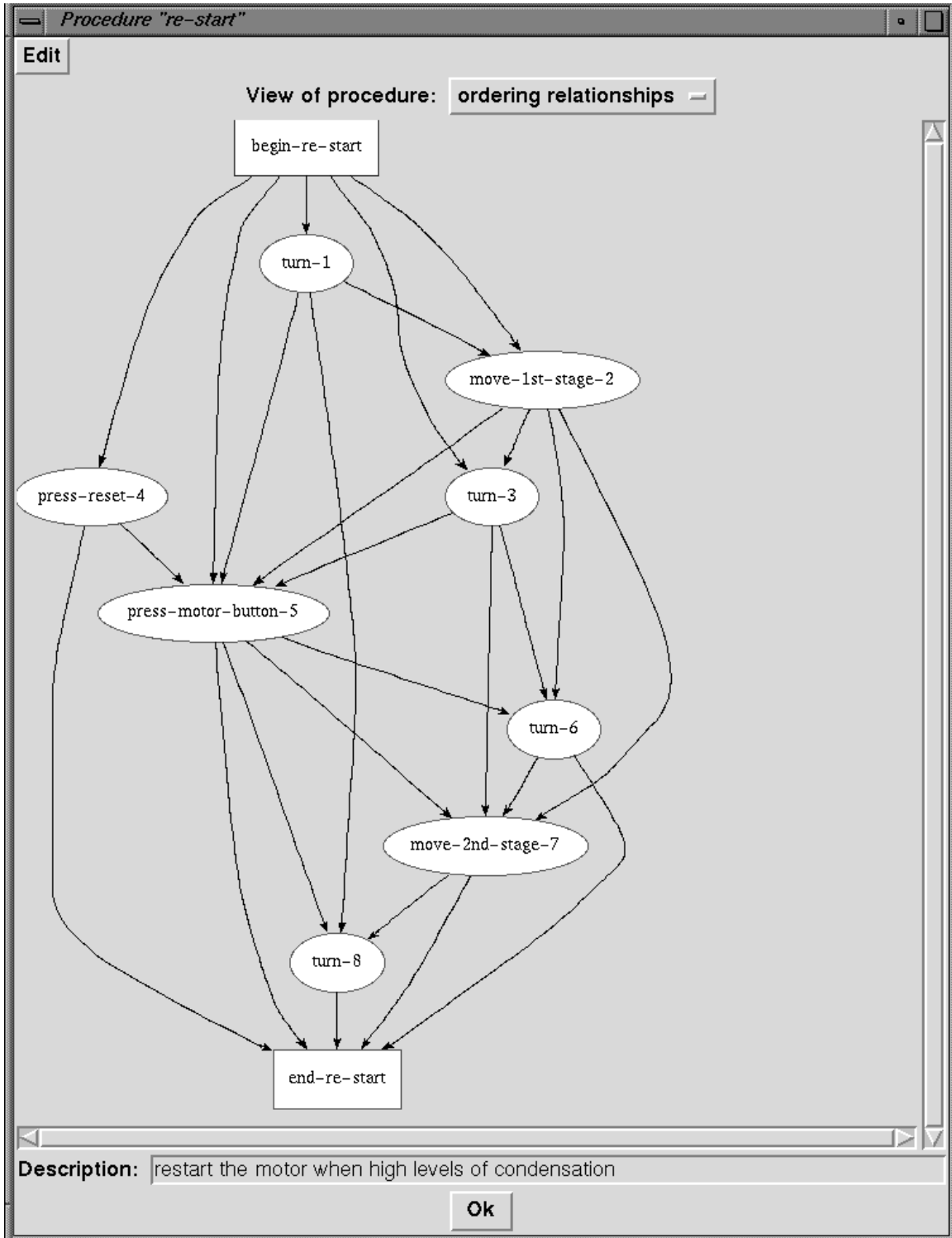


Figure 2.8: The Dependencies After Experimentation

can change the data shown on menus, and Diligent's implementation does not support dynamically updating menus with the results of experiments.

At this point, let us suppose that Diligent experiments with our example procedure. The improved procedure would now look like Figure 2.8. The experiment allowed Diligent to remove unnecessary dependencies, such as the one between steps **turn-1** and **turn-3**.

2.3.2.7 Additional Authoring Activities

At this point, the instructor could refine the example by performing additional authoring activities. He could use menus to edit the procedure, he could test the procedure with the STEVE tutor, and he could provide Diligent with additional demonstrations. Additional demonstrations can be used to add new steps to an existing procedure, but demonstrations do not have to add steps. Diligent also supports demonstrations that are only used to help learn operator preconditions. However, demonstrations that focus on learning preconditions without adding steps would probably only be used by an expert user.

Diligent allows editing, testing and demonstrating to be interleaved and repeated. This allows an instructor to iteratively refine a procedure.

2.4 Summary

This chapter presented a high-level discussion of the basic concepts involved in using Diligent.

We first discussed a simple procedure for turning on a car's motor. This example was used to identify properties of procedures. The purpose (or goal) of a procedure is to make changes to the world. This is done by performing a sequence of steps where some later steps may be dependent on changes caused by earlier steps.

We then discussed what concepts are needed for simple use of Diligent. Users need to understand the basic representation (but not operators). Users should also understand that Diligent uses demonstrations and can improve its knowledge with experiments.

For more advanced use, a user should understand the operator representation and have some idea of what types of examples will improve learning. Users should also understand Diligent's heuristic assumption that the state changes of earlier steps are likely to be preconditions of some later steps. Users should also understand extensions to the basic representation. For example, an advanced user should understand that a large procedure could be composed hierarchically by using other procedures as steps in the large procedure.

Finally, we went through an example that illustrated what instructor would have to do in order to author a procedure.

Chapter 3

Diligent – Its Problem and Approach

As discussed previously, this thesis presents an approach for learning procedures for use in a heterogeneous, simulation-based tutoring system. In our approach, a human instructor authors procedures by demonstrating them in manner similar to how he would demonstrate them to a human student. This thesis focuses on understanding these demonstrations. The demonstrations are understood by first observing them and then refining the system's understanding with experiments.

The approach was implemented in a system called *Diligent*, which was used to learn procedures for machine maintenance. However, the approach is independent of any particular domain.

This chapter describes how Diligent works and the problem that it is addressing. First, we will characterize the problem. Second, we will discuss inputs and outputs. Third, we will describe the basic heuristics and how demonstrations are turned into procedures. The chapter finishes by indicating where various algorithms are discussed in greater detail.

3.1 The Problem

The problem that Diligent addresses can be described by high-level requirements, constraints on the nature of the domain knowledge, and the functionality provided by the interface to the simulated domain (or environment).

3.1.1 Requirements

Diligent has several high-level, general requirements.

Produce a procedure. Diligent needs to learn procedures that can be used for teaching human students.

Understand demonstrations. An instructor specifies the steps of procedures with demonstrations. Therefore, in order to understand procedures, Diligent needs to understand demonstrations.

3.1.1.1 Reducing the Instructor's Effort

Diligent has a general requirement to reduce the effort required from an instructor. This requirement has a number of aspects.

Maximize the utility of each demonstration. Not only do demonstrations require some effort on the part of an instructor, but the number of demonstrations that an instructor can provide is limited. Therefore, it behooves Diligent to extract as much information as possible from each demonstration.

Save time. A goal of a system like Diligent is to reduce the amount of time required by an instructor. One way to do this is to get the maximum use out of each demonstration.

Reduce the difficulty. Diligent should make authoring easier. We assume that easier authoring helps improve the quality of the resulting procedures because an instructor less likely to have a lapse of concentration and because he can focus his attention more effectively. Authoring should be easier if it takes less time and if the system gets the maximum use out of each demonstration.

Exploit the environment. Diligent should gather as much information as possible from the environment in order to avoid asking the instructor unnecessary questions. By exploiting the environment, Diligent can learn more from demonstrations and make the instructor's job less difficult while taking less time.

Provide aid when possible. Whenever possible, Diligent should aid the instructor by doing tasks that it can easily automate. For example, given knowledge of a procedure, Diligent can easily output the procedure in the form of a plan. However, given the same information, the instructor may have difficulty creating a plan.

3.1.2 Constraints on the Domain Knowledge

The available domain knowledge constrains which hypotheses Diligent proposes and how Diligent approaches learning.

Little initial knowledge. When Diligent starts, it may have no knowledge of the domain. Of course, Diligent gains knowledge as it interacts with the instructor, but Diligent still needs to be able to function with very little knowledge.

This thesis looks at general-purpose techniques that would allow an authoring tool to be easily ported to a different domain. For this to happen, little explicit encoding of domain knowledge should be required from users. Because little knowledge may be available, the techniques need to be robust enough to provide an instructor with “useful” feedback when only given a small amount of data. For example, Diligent should be able to identify likely dependencies between a procedure’s steps after only a single demonstration of that procedure.

Idiosyncratic Objects. In machine maintenance domains, every member of a class of objects (e.g. buttons or switches) may behave differently. This happens because they have different functions (e.g. start the motor versus turn off the lights). Thus, actions performed on the same class of object may have very different preconditions and produce very different state changes.

Because objects of the same class may behave so differently, Diligent makes no attempt to generalize an action’s preconditions so that the preconditions can be applied to actions on similar objects.¹

Unstructured environment. Diligent may have an unstructured environment. The state of *unstructured* environment only contains a set of attribute values. An unstructured environment contains no information about the relationships between attributes, between objects, or between attributes and objects.

Because structural knowledge might be easy to acquire, why doesn’t Diligent require it? First, Someone would have to explicitly encode the knowledge if it were unavailable. Second, because objects may have idiosyncratic behavior, it would be difficult to use structural knowledge to make generalizations that apply to a class of objects rather than to a specific object. Third, the machine maintenance domains that we have looked at contain a lot of constraints between objects, and many of these constraints are unknown. This means that an action on one object may have an effect on another object that appears to be totally unrelated to the first object.

¹We considered learning generalized preconditions that could apply to a class of object. If a particular object were to have unusual behavior, then the object’s behavior could have been modeled differently than the other objects in the class. While this approach is a possible extension, it did not appear very useful in the domains that we were using.

For example, pressing a button could drain an overflow tank in another room. Since structural knowledge is not of obvious benefit in our domains, not requiring it makes Diligent's techniques more general.

Can observe all relevant attributes. Diligent can see every attribute in the environment that is relevant. Relevant attributes include those that are used for teaching students as well as attributes that are required to make the actions performed in the environment appear deterministic. Because Diligent can see all relevant attributes, it does not need to detect or model missing attributes.

Most attributes are irrelevant. In a complex domain, there may be hundreds if not thousands of attributes. For example, the simulation in our Gas Turbine Engine domain internally used over 10,000 attributes. In Diligent's main domain (High Pressure Air Compressor), the simulation internally used approximately 4,500 attributes, of which approximately 70 appeared to have potential use in teaching.² However, for any given procedure, most attributes are very likely to be irrelevant. This means that the learning algorithm will need to focus on filtering out irrelevant attributes.³

3.1.3 Interface with the Environment

In this document, the parts of the heterogeneous tutoring system that simulate the domain are called the *environment*. Because this work focuses on using the environment, we need to discuss the functionality provided to Diligent by the environment.

Diligent places very few demands on the interface to the environment, and because the interface is very general, this interface could be supported by a wide variety of simulations.

Because our purpose is to convey the basic functionality provided by the environment, we will use informally defined data types. These data types will be defined formally in Section 3.2.1.1.

```
procedure Current-State  
  output: state-vector
```

The procedure **Current-State** returns a state-vector, which contains the current values of the attributes that Diligent can observe. This functionality is necessary when an

²Diligent and the STEVE tutor had access to approximately 40 attributes.

³Identifying attributes that are relevant or likely to be relevant is a major focus of the algorithms discussed in the chapters on learning operators and experimentation.

automated tutor interacts with human students. It is also useful when an instructor tests a procedure.

```
procedure Observe-Action  
  output: action-example
```

The procedure **Observe-Action** allows Diligent to observe the instructor as he performs a demonstration. The action-example identifies which action the instructor performed and how it changed the state of the environment. The change in state is represented by the environment's state before and after the instructor performed the action.

It is assumed that Diligent can see all actions performed by the instructor and that Diligent is notified whenever an action is performed.

```
procedure Perform-Action  
  input:  action-id  
  output: action-example
```

The procedure **Perform-Action** allows Diligent to perform experiments. The action-id identifies which action should be performed (e.g. press the red button). The action-example identifies the environment's state before and after the action was performed.

```
procedure Restore-Environment-State  
  input:  configuration-id
```

The procedure **Restore-Environment-State** allows Diligent to restore the environment to a known state. The configuration-id is a text string that the instructor provides, and it identifies a known configuration of the environment. Other than this text string, Diligent has no knowledge about what data is required to restore the environment's state.

The ability to reset the environment is reasonable in a system that tutors human students. This ability allows students to start working on procedures from known, pre-specified initial states.

Diligent uses **Restore-Environment-State** before starting a demonstration. This allows Diligent to later restore the demonstration's initial state so that Diligent can perform experiments or the instructor can provide additional demonstrations.

One procedure that is not listed is **Save-Environment-State**. While instructors require this capability in order to create configurations for **Restore-Environment-State**, Diligent does not dynamical create new environment configurations because it takes too

long in the tutoring environment in which Diligent was developed. To see why saving the environment's state could take a long time, consider that a simulation could use thousands of attributes, and the values of each of these attributes would need to be saved.

As will be discussed later, Diligent gets around its inability to save the environment's state by remembering an existing configuration and the subsequent actions used to modify the environment's state.

Besides the functionality to observe and manipulate the environment, Diligent requires some functionality to map the identifiers used by the environment to English descriptions. This has a couple of advantages in a heterogeneous tutoring system. First, this allows different components of the system to use the same terms when discussing the same objects. Second, by having a centralized repository of descriptions, there is less likely to be problems because of inconsistencies or errors in redundant descriptions.

procedure **Action-Type-Description**

input: action-type
output: description

The procedure **Action-Type-Description** describes the type of action that the instructor performed. Some examples are “pressing” a button or “toggling” a switch.

procedure **Object-Description**

input: environment-object
output: description

The procedure **Object-Description** provides a name for domain objects (e.g. “reset button” or “alarm light”).

procedure **Attribute-Description**

input: attribute-name
output: description

The procedure **Attribute-Description** describes attributes of the environment. Attributes can represent things such as whether a valve is open or whether a light is illuminated. For example, a description of attribute `valve1` might be “first stage value.”

There is no procedure **Attribute-Value-Description** because Diligent assumes that attribute values are English text strings that can be understood by students. For example, the state of a light might “on,” or the position of a valve might “off.”

One consequence of this relatively limited interface is the following constraint.

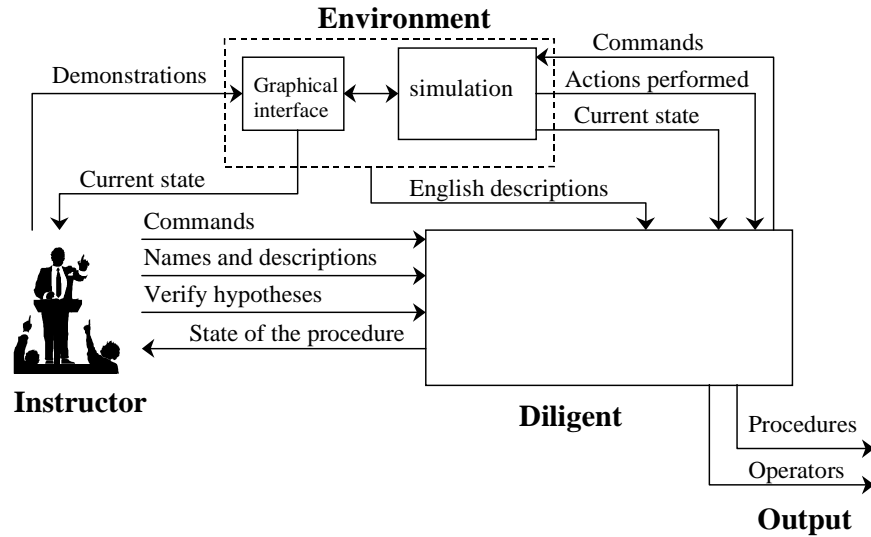


Figure 3.1: Input/Output

Cannot directly change attribute values. Although Diligent has the ability to restore the environment’s entire state, we do not assume that Diligent has the ability to independently set the value of individual attributes (e.g. set the value of `valve1` to `open`).

This restriction might be unnecessary in domains with few constraints on valid attribute values, but in some domains (e.g. machine maintenance domains), there tend to be a large number of constraints. Consider a simulation of an automobile. One source of constraints is consistency between attribute values. For example, depressing the gas pedal causes more gasoline to flow to the engine. The simulation would be in an inconsistent state if Diligent was allowed increase the depression of the pedal without increasing the flow of gasoline. Another source of constraints is avoiding states that the environment does not support. For example, the simulation may not correctly model the behavior of the automobile’s suspension at high speeds. If Diligent were able to change the automobile’s speed to a high value, then the behavior of the simulation would be invalid.

3.2 Input and Output

Figure 3.1 shows the input and output involved in authoring procedures. In the figure, the lines represent data flows and the rectangles represent programs. While Diligent and the instructor treat the environment as a single entity, the environment actually consists of a graphical interface and a simulation. The simulation controls the state of the graphical interface. The instructor interacts with the environment by manipulating and examining the graphical interface, while Diligent interacts directly with the simulation.⁴

3.2.1 Input

When learning a procedure, Diligent gathers input from both the instructor and the environment.

The instructor uses the environment's graphical interface to demonstrate a procedure. Diligent observes the demonstration as a sequence of action-examples that it receives from the environment. (As will be described below, an action-example indicates what action was performed and the environment's state before and after the action.)

As the instructor works on a procedure, he can examine the state of the environment by looking at the graphical interface. However, like the "real-world", the instructor may be unable to directly observe the values of some of the environment's attributes (e.g. whether the internal pressure is normal).

The instructor uses Diligent's menus to provide additional knowledge beyond demonstrations. He provides text strings for names and descriptions. For example, a procedure might be called "procA" and have the description "shut down the device." He can also verify Diligent's hypotheses by making discrete choices. A discrete choice may involve indicating "yes" or "continue" (e.g. accepting hypothesized procedure goals), or it may involve selecting one of several possibilities (e.g. rejecting a precondition). Additionally, the instructor controls the interaction with Diligent through commands (e.g. start a demonstration).

Besides demonstrations, the environment also provides Diligent with English description of attributes, objects and actions. One use of these descriptions is mapping the environment's internal representation to something that the instructor can understand. Another use is building default descriptions that could be used with human students.

⁴The simulation is implemented with VIVIDS, which is a version of the University of Southern California's Behavior Technology Laboratory's RIDES [MJP⁺97], and the graphical interface is implemented with Lockheed Martin's Vista Viewer [SMP95]. Additional details of the implementation are located in Appendix A.

Diligent also has the ability to send two types of commands to the environment. One type resets the environment to a known state. This allows Diligent to setup and restore a demonstration’s initial state. Besides restoring the state, Diligent can also perform the same types of actions (e.g. pressing a button) as the instructor. This ability allows Diligent to experiment.

3.2.1.1 Definitions of Basic Data Types

identifier	→	string of ASCII characters
description	→	string of ASCII characters that a human student should be able to understand
attribute-name, environment-object, action-type, configuration-id	→	identifier
attribute-value	→	description
condition	→	(attribute-name attribute-value)
action-id	→	action-type environment-object
state-vector, partial-state	→	set of conditions
pre-state, post-state	→	state-vector
delta-state	→	partial-state
action-example	→	action-id pre-state post-state delta-state

Figure 3.2: Syntax of Basic Data Types

Figure 3.2 shows the syntax of the basic data types used for input.

Two types of ASCII strings are used: identifiers and descriptions. *Descriptions* contain text that a human student should be able to understand. For example, an *attribute-value* is a description and should be something like “open” or “on” rather than something like “x25” or “LtPurp.”⁵ In contrast, *identifiers* are not given to students. Because Diligent uses some identifiers internally, an instructor may only be aware of a few types of identifiers (i.e. attribute-name and configuration-id).

The interface uses identifiers for a variety of purposes. An *attribute-name* identifies an attribute in the environment. An *environment-object* is an object in the environment that can be manipulated (e.g. reset button). An *action-type* indicates what type of action is

⁵If attribute values could not be understood by students, the environment would have to provide Diligent with a means of translating a value into a human understandable form.

performed (e.g. pressing a button versus toggling a switch). A *configuration-id* identifies an internal state of the environment. A configuration-id allows Diligent and the instructor to communicate about known states. Configuration-ids are also needed because Diligent may only be aware of a small fraction of the environment’s internal state.

The portion of the environment’s state that Diligent can see is represented as a conjunction of conditions. A *condition* contains an attribute and its value, and represents an equality relation. A condition is said to be *satisfied* or *true* if the attribute has the given value; otherwise, the condition is said to be *unsatisfied* or *false*. For example, the condition (**valve1 open**) indicates that attribute **valve1** has the value **open**.

Diligent only supports conditions that involve an equality relation between an attribute and a small set of discrete values (e.g. “solid,” “liquid,” or “gas”). Attribute values cannot be continuous (e.g. real numbers), and the relation between an attribute and its value cannot be a relation such a “less than,” “greater than,” or “not equal.” For example, a condition cannot represent the relation “temperature < 100 degrees.”⁶

An instructor performs actions in the environment by selecting objects in the graphical interface with the mouse. An action is represented with an *action-id*. An action-id identifies the type of action (e.g. pressing a button) and the object acted upon (e.g. the reset button). (How Diligent uses action-ids will be discussed below when we cover operators, which are used to model actions.)

While Diligent could have allowed an instructor to perform an action by using a menu to select from a list of actions, Diligent does not assume that it knows all possible types of actions or all objects that could be acted upon.⁷

The interface with the environment uses sets of conjunctive conditions. One type of set is a *state-vector*, which contains all attributes that Diligent can see. Another type of set is a *partial-state*, which may contain zero or more conditions.

An *action-example* indicates how an action (*action-id*) affected the environment. The state before the action is called the *pre-state*, and the state afterwards is called the *post-state*. The *delta-state* contains the post-state conditions of attributes that changed value during the action.

To support identification of the pre-state and post-state, it assumed that actions are discrete rather than continuous. This means that there is an identifiable time before an

⁶Relaxing the assumption that all conditions represent equality relations is briefly discussed in Section 8.2.

⁷When we evaluated Diligent, we created a version of Diligent that allowed users to specify an action by using a menu to select the action from a list of predefined actions.

action starts and after it finishes. Actions that have delayed effects, which appear after subsequent actions have been performed, are beyond the scope of this thesis and are an area for future work.

Furthermore, it is assumed that actions are deterministic in that an action will produce the same post-state when performed in the same pre-state.

```
action-id: turn handle1 (i.e. turn the handle on a valve)
pre-state:
  (valve1 open)(valve2 open)(HandleOn valve2)
post-state:
  (valve1 open)(valve2 shut)(HandleOn valve2)
delta-state:
  (valve2 shut)
```

Figure 3.3: An Action-Example

Figure 3.3 shows an action-example. In the example, an instructor selects a handle that is used to open or shut valves. When the instructor does this, the valve underneath the handle (`valve2`) is shut. The only change in the state (i.e. delta-state) is that `valve2` is now `shut`.

3.2.2 Output

Diligent produces two types of objects (i.e. procedures and operators) that could be given to an automated tutor. Procedures describe how to perform tasks, such as restarting an air compressor, and operators model how the actions performed by an instructor affect the state of the environment. While only procedures can be directly used for teaching, a tutor could use operators for more robust error recovery and for modifying existing procedures.⁸

To see how operators could help a tutor, consider the following example. Suppose a student is trying learn how to checkout an air compressor, and he mistakenly turns off the compressor's power. The procedure used by the tutor should allow the tutor to deal with most unexpected events and student errors, but what happens if the student performs an action that is not described by the procedure. If the tutor only used the procedure, it might not be able to provide the student with help, but if the tutor used operators, it might be able to modify the procedure so that it could provide help.

⁸Extensions to the basic procedure and operator representation are discussed in Chapter 8.

3.2.2.1 Procedures

As was discussed in the first chapter, Diligent outputs procedures in a form (i.e. hierarchical partially ordered plans [RN95]) that is commonly used by the Artificial Intelligence community and has often been used in research on explaining procedures to humans. This research on explaining procedures to humans has looked at topics such as providing concise descriptions [You97], selecting rhetorical relations to express the relationships between actions [VM95, Van93], multilingual instruction generation [DHP⁺94, PV96, PVF⁺95], and representing relations that hold between pairs of actions [Pol90, Di 94, Bal93].

Research has identified features of this representation that are needed to provide adequate explanations [ME89, You97].

- Large, complicated procedures can be decomposed into a sequence of smaller, simpler and logically coherent subprocedures. For example, consider a procedure that teaches someone to drive to the store. One subprocedure might involve starting a car, and another subprocedure might involve stopping at a stop light.
- The representation describes causal dependencies between steps that can be used to answer questions about how to perform a procedure or why is an action needed.⁹

To clarify the discussion, we will use the example plan in Figure 3.4.

The components of a procedure that Diligent outputs include

- *Name*. Names are used with an instructor to identify a procedure. Diligent’s implementation requires each that procedure have a distinct name. In the example, the plan is named **procA**.
- *Set of steps*. Each step corresponds to an action performed in the environment or to another procedure. A procedure embedded inside another procedure as a step is called a *subprocedure*, and the procedure containing a subprocedure is called the *parent* procedure. Steps representing a subprocedure are called *abstract*, while steps representing an action are called *primitive*. For implementation reasons, procedures are trees in that no procedure can recursively be a step inside itself or any of its subprocedures.

⁹Questions about how to perform a procedure and why to perform a step were briefly discussed in Section 1.1.2. However, this topic is not a focus of this dissertation.

Name: procA

Steps:

begin-procA, turn-1, press-system-test-2, end-procA

Goal conditions:

(valve1 shut)(CdmStatus test)

Causal links:

begin-procA	establishes (HandleOn valve1)	for turn-1
begin-procA	establishes (valve1 open)	for turn-1
begin-procA	establishes (CdmStatus normal)	for press-system-test-2
turn-1	establishes (valve1 shut)	for press-system-test-2
turn-1	establishes (valve1 shut)	for end-procA
press-system-test-2	establishes (CdmStatus test)	for end-procA

Ordering constraints:

turn-1 before test-system-test-2

Figure 3.4: Example Plan `procA`

For implementation reasons, each step has to have a distinct name. To prevent two steps from having the same name, most steps have a number appended to their name (e.g. `turn-1` and `press-system-test-2`).

To simplify processing, each procedure has two additional steps that represent the procedure's initial and final state (e.g. `begin-procA` and `end-procA`). Because the names of these steps are created using the procedure's name (e.g. `procA`), the step names are already distinct and do not need a number appended to them.

- *Set of goal conditions.* The purpose of the procedure is to establish a conjunctive set of goal conditions. The procedure terminates when all goal conditions are satisfied. When the procedure terminates, the environment is said to be in the *goal state*. The example procedure has the goal condition (`valve1 shut`). This means that `valve1` needs to be **shut** at the end of the procedure.
- *Set of causal links.* A causal link [MR91] indicates that performing a step causes a precondition for another step to be true. The example procedure has the causal link `turn-1 establishes (valve1 shut) for press-system-test-2`. This means that step `turn-1` shuts `valve1` and that `valve1` being shut is a precondition of step `press-system-test-2`.

For bookkeeping purposes, there are also causal links involving the initial and goal states. In the example, conditions that are part of procedure’s initial state are identified with causal links involving the step **begin-procA**. Conditions that are part of the goal state are identified with causal links involving the step **end-procA**.

Because causal links identify fine-grained dependencies between steps, they have been found to be very useful when describing procedures to humans [ME89].

- *Set of ordering constraints.* An ordering constraint indicates the relative order for performing a pair of steps. For example, the ordering constraint **turn-1 before test-system-test-2** indicates that step **turn-1** should be performed before step **test-system-test-2**.

By definition, all steps are performed after the start of the procedure (i.e. **begin-procA**) and before the end of the procedure (i.e. **end-procA**).

- *Text descriptions.* The procedure also contains English descriptions that can be given to human students. There are descriptions for the procedure, individual steps and causal links.

In this document, the term *step relationships* is used to describe both causal links and ordering constraints.¹⁰

Throughout this thesis, plans will be represented in the format shown in Figure 3.4. The order of steps, causal links, and ordering constraints is meant to improve readability and is not important. Steps are listed in the order that the instructor demonstrated them. Causal links and ordering constraints are ordered so that those involving the procedure’s earlier steps are listed before those involving later steps.

3.2.2.2 Operators

Besides procedures, Diligent also learns operators. Although operators are not part of a plan, Diligent internally associates one operator with each primitive step. Diligent uses a step’s operator to identify the step’s preconditions, which are used when generating the plan’s step relationships.

Diligent uses operators to model reusable knowledge of how an action affects the environment. Because operators model the environment, they are not specific to a given step

¹⁰Diligent’s implementation and its training documentation used the term *ordering relationships* instead of *step relationships*. This document uses the term *step relationships* so that the reader does not confuse ordering relationships with ordering constraints.

Name: toggle-motor

Action-Id: press motor-button

Effect: effect1

Preconditions:

g-rep:

(motor on)

h-rep:

(motor on)(valve1 open)

s-rep:

(motor on)(valve1 open)(valve2 open)(HandleOn valve1)

State changes:

(motor off)

Effect: effect2

Preconditions:

g-rep:

(motor off)

h-rep:

(motor off)(valve1 closed)

s-rep:

(motor off)(valve1 closed)(valve2 open)(HandleOn valve1)

State changes:

(motor on)

Figure 3.5: Example Operator `toggle-motor`

or procedure. Operators model an action by identifying the set of conditions (or preconditions) necessary so that performing the action will achieve specified state changes. A state change indicates that an action caused an attribute to change value and is represented by a condition that contains the attribute's new value.¹¹

An operator consists of the following components. To clarify the discussion, we will use the example operator in Figure 3.5.

- *Name*. The name used to identify the operator to the instructor. For implementation reasons, Diligent allows operators to have duplicate names. In the example, the operator is called **toggle-motor**.
- *Action-id*. The action-id uniquely identifies the operator. The operator in the example models pressing the **motor-button**.

An operator is only associated with one action-id. Diligent treats action-ids as atomic and does not compare action-ids to look for common objects or common types of actions. As was discussed in Section 3.1.2, operators do not model same the type of action on multiple objects (e.g. all buttons) because different objects of the same type (e.g. motor button versus reset button) may affect the environment in very different ways. Similarly, different types of actions on the same object are modeled independently because the actions may perform totally different activities (e.g. “pressing” the button versus “removing” it).

- *Description*. The description provides a default English description for the plan steps associated with the operator.
- *Conditional effects (or effects)*. In different situations, performing an action can produce different state changes. These differences in behavior are modeled with effects. Each effect identifies the preconditions necessary for the action to produce specific state changes. The example operator has two effects: one effect models turning on the motor, and the other effect models turning off the motor. Each effect consists of the following components.
 - *One or more state changes*. State changes identify how the action changes the state of the environment. Each state change identifies an attribute whose value is changed by the action. A state change is is represented by a condition that

¹¹Chapter 5 discusses operators and motivates their representation in greater detail.

contains the attribute’s new value. The example’s **effect1** has the state change (**motor off**), which means that the motor is turned off.

Because a state change models changes to the environment’s state, each effect has implicit preconditions that the attributes in its state changes cannot have their post-action (or post-state) value before the action takes place. For example, **effect1**’s state change (**motor off**) gives **effect1** the implicit precondition that the motor is *not* turned off before the action is performed.

- *Preconditions.* For an action to produce an effect’s state changes, the effect’s preconditions should be true. An effect has three sets of conjunctive preconditions: g-rep, h-rep and s-rep.

The g-rep and s-rep represent a version space [Mit78]. The version space bounds the correct precondition between the most general candidate precondition (*g-rep*) that is consistent with the data and the most specific candidate precondition (*s-rep*) that is consistent with the data. The g-rep contains all preconditions that have been proven to be necessary, while the s-rep contains all potential preconditions that have not been proven to be unnecessary. For example, in **effect1**, the g-rep contains (**motor on**) because Diligent has proven that the motor needs to be turned on. In **effect1**, the s-rep contains all conditions ((**valve1 open**)(**valve2 open**)(**motor on**)(**HandleOn valve1**)) that Diligent has not proven to be unnecessary.

Because there may be little data for learning, the g-rep is likely to be too general, and the s-rep to be too specific. To overcome this lack of data, Diligent has a third set of preconditions, the *h-rep*, which represents Diligent’s heuristic, best guess about the “real” preconditions. The h-rep contains every condition in the g-rep and some conditions in the s-rep. For example, the h-rep for **effect1** contains two conditions. One condition has been show to necessary ((**motor on**)), while the other ((**valve1 open**)) is only likely to be necessary.

3.3 How Diligent Works

3.3.1 Processing Demonstrations

Figure 3.6 summarizes how Diligent learns procedures. The rectangles represent data objects and the ovals represent activities.

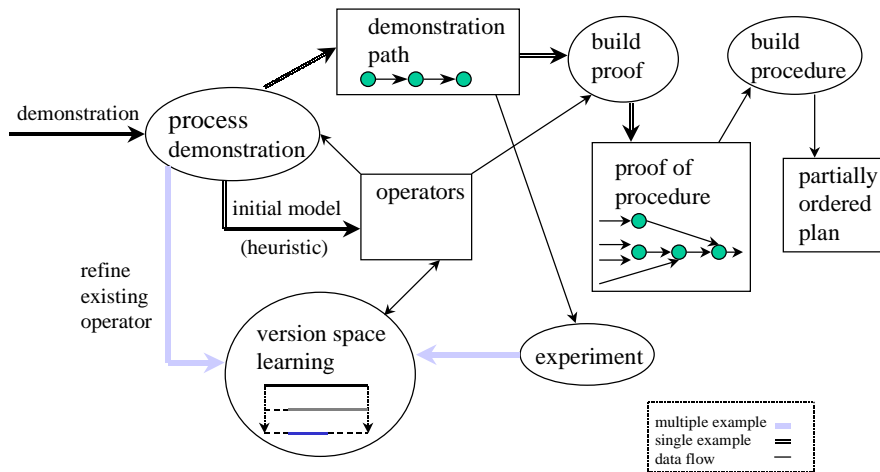


Figure 3.6: Processing a Demonstration

Initially, the instructor demonstrates the procedure. Diligent observes one action-example for each step in the demonstration, and it uses the action-examples to create a path for the demonstration. The *path* can be thought of as containing the demonstration's sequence of action-examples. The path does not indicate any causal relationships between the steps.

During a demonstration, Diligent creates and refines operators for each of the demonstration's steps. If Diligent has not seen a step's action before, it creates a new operator. Otherwise, an existing operator is refined. Diligent stores operator preconditions in a version space representation [Mit78]. The version space bounds the "real" precondition between a most specific and a most general candidate precondition. Because version space learning may converge slowly, Diligent has a heuristic precondition that is in between the most general and specific candidate preconditions. This heuristic precondition is used for generating plans.

Once a procedure has been demonstrated and operators are defined, the instructor can tell Diligent to generate a proof of the procedure. This data structure is called a *proof* because it records how the preconditions and state changes of the path's steps are used to transform the procedure's initial state into its goal state. Given a proof, it is trivial to transform the proof into a partially ordered plan.

However, if the operators are not very refined, a plan can have missing or unnecessary step relationships. The instructor can help correct this problem by telling Diligent to experiment. When experimenting, Diligent uses the environment to repeatedly perform the demonstration while skipping a step. Eliminating a step exposes how the missing step affects later steps. After the operators are more refined, Diligent can then generate a new version of the plan.

3.3.2 Heuristics

Diligent provides authors with heuristic aid when authoring procedures. The aid is “heuristic” because Diligent does not receive enough data to guarantee the correctness of what it learns. Instead of correctness, Diligent attempts to provide an instructor with a “reasonable” procedure.¹²

Most of the machine learning in this thesis focuses on learning operator preconditions. As mentioned earlier, in any procedure, most of the environment’s attributes are probably irrelevant. This means that Diligent’s learning algorithms need to understand demonstrations well enough to identify the subset of attributes that are likely to be used in preconditions. By identifying a “likely” set of preconditions, Diligent can make an instructor’s job easier because it allows him to focus on a smaller set of candidate preconditions.

One heuristic that Diligent uses to identify preconditions is the logical fallacy “*post hoc, ergo propter hoc*,” which means “after this, therefore because of this” [She97]. In other words, things that happened earlier in a demonstration are likely to cause things later in a demonstration. This is a fallacy because correlation does not equal causation.

The heuristic has two relevant aspects.

Earlier steps establish preconditions of later steps. The steps in a demonstration are related, and the instructor probably has reasons for demonstrating the steps in a given order. A likely reason for this ordering is that some state changes of earlier steps establish preconditions of later steps.

Focus on attributes that change value. Attributes with a constant value will not affect whether a demonstration’s final state is achieved when the demonstration’s steps are performed in a given order. In contrast, attributes that change value may differentiate between orders of steps that achieve the demonstration’s final state and those that don’t.

¹²By “reasonable” procedure, we mean that it should be close to correct.

Another way to look at this is to consider an action that produces different state changes when performed in two different states. Attributes with the same value in both states did not cause the difference in the state changes. In contrast, at least one difference between the states is a precondition.

Just because Diligent concentrates on attributes that change value does not mean that an attribute with a constant value is irrelevant – it just means that there is only weak evidence that the attribute is used in a precondition.

When Diligent has different levels of knowledge, different techniques be may appropriate. Initially, Diligent may have little domain knowledge. To compensate for its lack of knowledge, Diligent uses general heuristics. However, as Diligent gains more knowledge, the same heuristics may no longer be appropriate. To handle this situation, Diligent has a heuristic to deal with existing knowledge.

Favor existing knowledge. Diligent favors existing knowledge and hypotheses over knowledge derived from general heuristics. In this aspect, Diligent is very simple because it does not consider the quality of existing knowledge.

Large procedures tend to be hierarchical in that some of a procedure's steps represent subprocedures. Furthermore, a subprocedure may itself contain its own subprocedures. This nesting of subprocedures can cause the total number of steps in a hierarchical procedure and its subprocedures to become very large.

As procedures become larger, run-time overhead becomes more of a concern. For hierarchical procedures, Diligent has a heuristic to reduce run-time overhead by limiting the amount knowledge that its algorithms use.

Focus on the current procedure. When Diligent interacts with an instructor, interaction is focused on the steps of the current procedure rather than the steps in a subprocedure. In fact, a subprocedure is treated as just another step that has preconditions and establishes state changes.¹³

To simplify processing, Diligent assumes that the step relationships of subprocedures are correct. This assumption allows Diligent to ignore many of the internal details of subprocedures.

¹³When authoring, an instructor works on a current procedure, which he has explicitly selected. As will be explained in Chapter 4, an instructor can explicitly include subprocedures as steps in the current procedure. The instructor must explicitly specify all subprocedures because Diligent cannot automatically decompose a large procedure into subprocedures.

As will be explained later, this heuristic is used to reduce the overhead of deriving step relationships and performing experiments.

3.4 Where to Look for More Information

Chapter	Chapter Number	Topic
Processing Demonstrations	4	Interaction with the instructor
		Types of demonstrations
		Assumptions about demonstrations
		Derivation of goal conditions
		Derivation of step relationships
		Creating hierarchical procedures
		Generation of default English descriptions
Learning Operators	5	Operator creation
		Operator refinement
Experimenting	6	Generation and performance of experiments

Table 3.1: Where Topics are Covered

Table 3.1 shows where the algorithms discussed in this chapter are covered in more detail. The chapter on demonstrations (chapter 4) covers how an instructor interacts with Diligent to perform demonstrations and how the demonstrations are then turned into plans. The later chapters on learning operators (chapter 5) and experimentation (chapter 6) concentrate on machine learning rather than authoring.