Chapter 4

Processing Demonstrations

Demonstrations by human instructors are Diligent's primary source of input. Yet, a demonstration is not a procedure. A demonstration doesn't identify the procedure's goals or how the demonstration's steps depend on each other.

This chapter describes the processing involved in transforming demonstrations into procedures. The chapter addresses a number of issues:

- The interaction between the user (or instructor) and Diligent. This includes assumptions about how the instructor demonstrates.
- The algorithms used to transform demonstrations into procedures that Diligent can output.
- How to construct a hierarchical procedure out of other procedures. A *hierarchical* procedure is a procedure that contains another procedure as one of its steps.

This chapter focuses on the interaction between Diligent and the instructor. We will start by briefly discussing authoring with Diligent. We will then discuss the data structures used to record demonstrations. Afterwards, we will discuss how to demonstrate a simple procedure and generate a plan for it. We will then discuss how to construct hierarchical procedures. We will also discuss how to incorporate steps into a procedure that gather information without changing the state of the simulated domain (or environment). We will then discuss complexity, and finish the chapter with related work on basic Programming By Demonstration (PBD) techniques.

4.1 The Authoring Process

In Chapter 2, we discussed how an instructor could use Diligent to author a procedure. We will briefly review this material.

Authoring a procedure involves specifying the procedure's steps and making sure that Diligent understands the relationships between the steps. After creating a new procedure, an instructor provides demonstrations for the procedure. Demonstrations can identify the procedure's steps as well as provide data for learning the preconditions of steps. After the instructor has defined the procedure's steps, Diligent is able to heuristically derive goal conditions for the procedure and to perform experiments that attempt to identify the preconditions of steps. At some point after the goals have been specified, the instructor can tell Diligent to derive the dependency relationships (i.e. step relationships) between the procedure's steps. Whether Diligent derives goal conditions, derives step relationships, or experiments is controlled by the instructor. The instructor controls when Diligent experiments to prevent experiments initiated by Diligent from causing the instructor undesired delays. When the instructor is satisfied with the procedure, he can give it to an automated tutor where it can be tested.

In order to make authoring easier, Diligent allows instructors to perform many iterations of these activities.

4.2 Types of Demonstrations

So far we've treated one demonstration as if it were analogous to one procedure. While that is the expected case, Diligent allows multiple demonstrations to be associated with a single procedure. Diligent supports the following types of demonstrations.[1]

- *Add-step.* Add-step demonstrations add steps to a procedure. This type of demonstration is used when a procedure is created, and it can also be used to add additional steps to an existing procedure. Additional steps are added to a procedure by inserting the new demonstration's steps in between a pair of the procedure's existing steps.

  Besides augmenting existing procedures, the ability to perform additional demonstrations supports error recovery.[2]

[1]Section 8.4.2.1 discusses extending Diligent to support additional types of demonstrations. These types of demonstrations did not appear important for the types of procedures that we used, but it appears that they would be useful on more complicated procedures.

[2]An instructor might detect errors by using menus to look at dependencies between steps, or he might detect errors by testing a procedure by with an automated tutor.

- *Clarification.* This type of demonstration lets the instructor illustrate how the domain works without adding steps to the procedure. Instead of adding steps, clarification demonstrations provide more data for machine learning. Clarification demonstrations can be used to show what happens if the demonstration is not performed properly, and they can be used to provide additional correct, but slightly different, demonstrations of the procedure.

  In Chapter 2, it was mentioned that only an expert user was likely to use this type of demonstration.

Diligent differs from most Programming By Demonstration systems by not requiring multiple, correct demonstrations of a procedure. Diligent can do this because it has access to the environment, which contains an executable model of the domain. Access to an executable model allows Diligent to perform experiments that can reveal information that would normally be provided by additional demonstrations.

As will be discussed later, both types of demonstrations are used to generate experiments.

4.3 Data Structures

This section presents the data structures used to process demonstrations. This discussion assumes some knowledge of how procedures are represented as plans (Section 3.2.2.1). The data structures use the basic data types that were defined for the interface to the environment (Section 3.2.1.1). The data structures will be illustrated later as we discuss processing demonstrations.

4.3.1 Prefixes

Each demonstration starts in a particular initial state, and Diligent remembers how to restore this initial state. Diligent restores the initial state when performing experiments and when the instructor provides additional demonstrations of the procedure.

The data structure used to store an initial state is called a *prefix.* Prefixes have the following components:

- *Configuration.* A configuration is a text string (i.e. configuration-id) that is used by the instructor and Diligent to communicate about known states of the environment.

- *Additional-actions.* A sequence of actions (i.e. action-ids) that modify the state of the environment that is specified by the configuration.

  Additional actions are useful for a couple of reasons.

  - Additional-actions can reduce the need for creating additional configurations of the environment. Saving the state of the environment in order get a new configuration-id might be expensive: not only could it take a long time, but it could also use a lot of memory.
  - Besides reducing the cost of saving configurations, additional-actions are used when embedding one demonstration inside another demonstration. This is useful when adding steps to an existing procedure. It is also useful when defining a new subprocedure as a step in another procedure.

4.3.2 Demonstrations

Demonstrations are the major source of input that Diligent receives from the instructor. To demonstrate, an instructor needs to provide an initial state and use the environment's graphical interface to perform a series of actions.

A *demonstration* has the following components:

- *Prefix.* The prefix contains the information necessary to restore the environment to the demonstration's initial state.

- *Previous-step.* The previous-step is useful in an add-step demonstration that adds steps to an existing procedure. The previous-step is a step defined by a previous demonstration. (For a procedure's first demonstration, the previous-step is the step representing the procedure's initial state.) A new demonstration's steps will be inserted into the procedure between the previous-step and the step immediately after it.

- *Steps.* The sequence of steps that the instructor demonstrates. A step is either a subprocedure or an action performed in the environment. A step in a demonstration is the same as a step in procedure. How a step changes the environment is recorded in the action-example that was produced when the step was demonstrated.

- *Type.* As discussed in Section 4.2, Diligent supports two types of demonstrations: add-step and clarification. (The steps of a clarification demonstration are not added to the procedure, but are used when Diligent experiments.)

4.3.3 Paths

One problem with the demonstration data structure is that it can be awkward for Diligent to use. Not only can a procedure contain steps from several add-step demonstrations, but a demonstration also references a previous step outside itself.[3]

To simplify processing, demonstrations are converted into a data structure called a path. Once the instructor has finished a demonstration, Diligent adds the demonstration's data to a path and no longer uses the demonstration. A path is easier to use than a demonstration because, unlike a demonstration, a path does not reference any steps outside itself.[4]

A *path* contains the following components:

- *Prefix.* Specifies the procedure's initial state.
- *Steps.* The sequence of steps to be performed.
- *Generates-plan.* Yes or no. Should the path be used to generate a plan? A "no" indicates that the path represents a clarification demonstration.

4.3.4 Steps

In order to create a plan for a procedure, Diligent needs to identify the preconditions of steps and the state changes produced by steps.

To illustrate the data associated with a step, we will use the following example:

> Consider a step where a valve is opened. Suppose the environment allows the valve to be opened whenever the valve is shut, but in the procedure being learned, the valve should only be opened if the alarm light is illuminated.

A *step* contains the following components:

- *Name.* Each step has distinct name.
- *Type.* Abstract, primitive or special. An *abstract* step represents a subprocedure, and a *primitive* step represents an action performed by the instructor. A special step indicates either the beginning or the end of the procedure.

[3] If one considers the step that represents the start of a procedure (e.g. begin-procA) as outside a demonstration, then demonstrations always reference a step outside themselves.

[4] Diligent uses paths rather than demonstrations to perform experiments.

- *Subprocedure.* The name of the subprocedure performed by an abstract step. This field is empty if the step isn't abstract.

- *Operator.* The operator that models the action performed by a primitive step. This field is empty if the step isn't primitive.

  An *operator* models how an action changes the environment. An operator does this by identifying the preconditions necessary for the action to produce a set of state changes. Because operators can be reused in other procedures, an operator's preconditions are independent of the current procedure. In the above example, the operator would indicate that the valve can be opened whenever it is shut. However, the operator should not contain procedure specific preconditions such as requiring the alarm light to be illuminated.

- *Control-preconditions.* Control preconditions are procedure specific preconditions for performing the step. In the above example, a control precondition should indicate that the valve should not be opened unless the alarm light is illuminated. The precondition is needed because the environment allows the the valve to opened whenever it is shut rather than when the light is illuminated. It appears that control preconditions are likely to refer to indicators such as lights and gauges that humans look at for visual cues.

- *Mental-conditions.* A mental condition is a condition that contains a mental attribute, and a *mental attribute* is an attribute that is internal to Diligent rather than present in the environment.

  Diligent creates mental-conditions for sensing actions. A *sensing action* [AIS88, RN95] gathers information from the environment without changing the state of the environment. For example, a sensing action might involve checking to see whether a light is illuminated or checking the value of a gauge. A human student might perform a sensing action on a light by looking at the light or selecting it with a mouse.

  Diligent uses mental-conditions to guarantee that a step is performed. Diligent's heuristics do this by putting all mental-conditions into the procedure's goal conditions. (Of course, the instructor can reject these goal conditions.[5]) If a mental-condition were not in the procedure's goal conditions, then the mental-condition's step would only be performed if the step's changes to the environment's state were

[5]Because he is a domain expert, an instructor should be able to determine which goal conditions seem valid or reasonable.

needed to complete the procedure. For example, if a sensing action's mental-condition was not part of the goal conditions or preconditions of other steps, then the step would never be needed because it does not change the environment's state.

- *Action-example.* An action-example (Section 3.2.1.1) is an example of an action being performed and identifies the state before the step (*pre-state*) and after the step (*post-state*). The portion of the post-state that changed is called the *delta-state.*

  The action example associated with a step comes from the instructor's demonstration of the step.

4.3.5 Revisiting the Representation of Procedures

A procedure consists of the following components:

- *A plan.* Plans are discussed in Section 3.2.2.1. Diligent outputs a procedure in the form of a plan.

- *Set of paths.* Diligent uses paths to generate plans. Diligent only allows one of a procedure's paths to generate a plan. However, it would not be difficult to extend Diligent so that multiple paths can generate plans. The path that generates the plan contains every add-step demonstration, while each clarification demonstration has its own path. Each clarification demonstration has its own path because clarification demonstrations are meant to be used only for learning and may not correctly perform the procedure.

  When Diligent experiments on a procedure, Diligent uses the procedure's paths to generate experiments. This includes paths that generate plans and those that don't.

4.4 Assumptions about How the Instructor Demonstrates

Before presenting some example demonstrations, we will discuss the nature of the demonstrations presented to Diligent. Diligent makes the following assumptions about demonstrations.

Correct demonstrations. Diligent assumes that an instructor knows how to correctly demonstrate a procedure. Diligent uses this assumption when it assumes that a path's sequence of steps is correct. Diligent also uses this assumption when it uses the action-example from the demonstration of a step.

Diligent allows an instructor to recover from a violation of this assumption by providing additional demonstrations, editing preconditions, and in the worst case, deleting a step and demonstrating it again.

Small, modular procedures. Diligent assumes that the instructor breaks large procedures into sets of small modular procedures. The instructor then uses the small procedures to construct large procedures. This assumption is used when considering the run-time overhead of some algorithms used to perform experiments or create plans.

First demonstration contains all steps. Because Diligent assumes correct demonstrations of small modular procedures, Diligent assumes that the first demonstration of a procedure probably contains all the procedure's steps. This assumption is used by Diligent when it considers only the current demonstration when creating the preconditions of a new operator. Because of this assumption, we did not focus on undesirable interactions between steps in different demonstrations of the same procedure.

Because this assumption is not always correct, Diligent allows the instructor to add steps to a procedure with additional demonstrations.

Multiple Demonstrations are Consistent. Suppose a procedure has multiple add-step demonstrations. Diligent assumes that the steps of a new demonstration will not remove preconditions that are required by steps later in the path. This assumption allows Diligent to use the action-example from a step's demonstration even though newer demonstrations may have changed the pre-state in which the step will be performed.

We did not focus on violations of this assumption because the procedures that we were looking at did not seem to require multiple demonstrations. This made to it difficult to find typical examples of how users would inconsistently perform multiple demonstrations. Instead, we focused on understanding small, modular procedures that were correctly demonstrated on the first demonstration.

Recovery from a violation of this assumption is similar to to recovery from an incorrect demonstration. Maintaining consistency of action-examples between demonstrations is an area for future work.

Logically related steps grouped together. Diligent assumes that the instructor groups logically related steps together in the same small procedure.

Violating this assumption not only causes the problems associated with misleading demonstrations, but also raises questions about whether the procedure being learned is usable. To some degree, Diligent's plans assume that students will finish one subprocedure before starting on the next subprocedure. When deriving a plan's step relationships, Diligent does not consider what would happen if the steps of two subprocedures were interleaved.

It is unclear whether authoring interleaved subprocedures is of any relevance. When an instructor provides demonstrations to Diligent, all subprocedures are performed sequentially, and thus, it is impossible to provide a demonstration with interleaved subprocedures. In any case, interleaved subprocedures are beyond Diligent's scope.

4.5 About this Chapter's Extended Example

This chapter's extended example shows how to author procedures by providing a series of demonstrations.

The demonstrations will be performed on a device called the High Pressure Air Compressor (HPAC). To improve clarity of presentation, the domain has been simplified by reducing the number of attributes and changing the names of attributes.

This chapter will not discuss the details of Diligent's user interface, which can be found in Chapter 2 and Appendix D.

4.6 Authoring a New Procedure

Initially, we will assume that Diligent has no knowledge of other procedures or about the domain.

4.6.1 Creating a New Procedure

The first thing that the instructor does is to create a new procedure. When creating a procedure, the instructor needs to give it a name and to provide a description. The name identifies the procedure to the instructor, and the description is used to describe the procedure to students. The instructor calls the procedure "proc1" and gives it the description "shut a few valves."

4.6.2 Setting Up the Initial State

Before demonstrating the new procedure, the instructor needs to put the environment into the demonstration's initial state. Since Diligent remembers this initial state, Diligent can restore the initial state for experiments and additional demonstrations.

Diligent asks the instructor for a configuration-id that identifies a known state of the environment. A configuration-id is a text string that Diligent and instructor use for communication. Let the instructor specify the configuration with the string "config1." Diligent then uses "config1" to reset the state of the environment with **Restore-Environment-State** (Section 3.1.3).

Because creating a new configuration of the environment may be slow or use a lot of memory, the instructor may wish to reuse an existing configuration while modifying the state associated with the configuration. For this reason, Diligent now asks the instructor if he'd like to modify the configuration "config1" by performing some additional actions. In our case, the instructor indicates that he doesn't want to perform any additional actions.

4.6.3 Demonstrating the Procedure

Now that the environment is in the desired initial state, the instructor can demonstrate the procedure. The demonstration contains three steps, and its purpose is to shut two valves. The steps are as follows.

1. The instructor uses the mouse to select the handle (handle1) that opens and shuts valves. This causes the handle to turn, which shuts the valve (valve1) that is underneath the handle.

2. The instructor moves handle handle1 to valve valve2 by selecting valve2 with the mouse.

3. The instructor selects handle handle1, which shuts valve2.

The instructor then indicates that he has finished the demonstration.

4.6.4 Creating Primitive Steps

In the above demonstration, none of the steps represents performing a subprocedure. Instead, every step represents an action that the instructor performs. When a step represents an action performed by the instructor, the step is called *primitive*.

Action-example: example1
  Action-id: turn handle1
  Pre-state:
    (valve1 open)(valve2 open)(HandleOn valve1)(AlarmLight1 off)
    (CdmStatus normal)
  Delta-state:
    (valve1 shut)

Action-example: example2
  Action-id: move valve2
  Pre-state:
    (valve1 shut)(valve2 open)(HandleOn valve1)(AlarmLight1 off)
    (CdmStatus normal)
  Delta-state:
    (HandleOn valve2)

Action-example: example3
  Action-id: turn handle1
  Pre-state:
    (valve1 shut)(valve2 open)(HandleOn valve2)(AlarmLight1 off)
    (CdmStatus normal)
  Delta-state:
    (valve2 shut)

Figure 4.1: First Demonstration's Action-Examples

Diligent gets information about how an action affects the environment in the form of action-examples. The action-examples for the demonstration's three steps are shown in Figure 4.1 and are used by **Create-Primitive-Step** (Figure 4.2) to create steps.

For the demonstration's first step, the instructor turns handle handle1. Diligent uses **Observe-Action** (line 1 in 4.2) to get the first step's action-example (example1). The action-example's action-id identifies which action was performed by indicating the type of action (turn) and the object being acted upon (handle1).

Diligent models how an action affects the environment with operators. Operators represent reusable, procedure-independent knowledge of the environment and can be used with multiple steps.

In order to reuse existing knowledge, Diligent searches for an existing operator that matches the action-id of the step's action-example. Diligent searches with the action-id because there is a one-to-one correspondence between operators and action-ids.

```
procedure **Create-Primitive-Step**

Input:    demo: The current demonstration.
Result:   stp : A step in the procedure.

1. Get the step's action-example ex from the environment
   with Observe-Action (Section 3.1.3).

2. Find if an operator op already exists for action-id(ex).

3. If an operator was found, refine op using the
   action-example ex. (Chapter 5).

4. Otherwise, no operator was found. (Need to create a new operator)

     5. Ask the user for an operator name and description.
     6. Use the operator name and description to create a new
        operator op. Creating the new operator requires
        the action-example ex and the current demonstration
        demo. (demo is used to create heuristic preconditions.)
        (Chapter 5).

7. Initialize the components of the new step stp. (The
   < integer > is used to give the step a distinct name.)

     name(stp) ← concatenate: name(op) "-" < integer >
     description(stp) ← description(op)
     action-example(stp) ← ex
     operator(stp) ← op

8. If the step represents a sensing action then initialize the
   the components of the step involving control preconditions and
   mental attributes. (Section 4.7.4.)
```

Figure 4.2: Creating a Primitive Step

At this point, Diligent doesn't know any operators. Therefore, Diligent needs to create an operator. First, Diligent asks the instructor to give the operator a name and a description. The instructor names the operator "turn" and approves the default description that Diligent has generated, "turn the valve handle."[6] Once the operator has a name and a description, the action-example and the current demonstration are used to initialize the new operator.[7]

Now that the step has an operator, Diligent uses the operator to create a name for the step. Since an operator could be used multiple times in a procedure, each step has a distinct name. The first step is called turn-1 and inherits the operator's description.

The last thing to do when creating a primitive step is to check whether it represents a sensing action (line 8 in Figure 4.2). A *sensing* action (e.g. checking a light) gathers information from the environment without changing it. Line 8 is skipped because none of the steps in this demonstration involve sensing actions.

For the second step, the instructor selects valve2. This moves the handle from valve1 to valve2. The step's action-example is example2. Once again, a new operator is created. The instructor calls the operator "move-2nd" and approves the default description "move to the second stage valve." This results in a step called move-2nd-2.

The operator is called move-2nd rather than move because different operators are needed to move the handle to each valve. An operator only models actions performed on one object and moving to a valve involves selecting that valve. As far as Diligent can observe, the only commonality in moving the handle to different values involves the type of action (move) and the attribute (HandleOn) whose value is changed. The problem is more difficult than it appears because the values of attribute HandleOn are actually descriptions of a valve rather than the name of a valve (e.g. "separator drain 1st stage valve" versus "valve1"). However, for clarity, we will use valve names (e.g. "valve1") as values of attribute HandleOn.

For the third step (turn-3), the instructor selects the handle again, which now shuts valve2. The step's action-example is example3. Unlike the first step, Diligent finds an operator (i.e. turn) that matches the action. Diligent then uses the step's action-example to refine the operator (line 3).

[6]The generation of default descriptions is described in Section 4.8.1.1.

[7]See Chapter 5 for details of how operators are created and then later refined.

4.6.5 Converting the Demonstration into a Path

As the instructor performs the new procedure's first demonstration, Diligent records it in the data structure shown in Figure 4.3. The type of demonstration is add-step because the demonstration adds steps to the procedure. Because this is the procedure's first demonstration, the previous-step (begin-proc1) represents the start of the procedure. The prefix records how to the demonstration's initial state was created and allows the initial state to be restored.

To make other processing easier, the demonstration is converted into a data structure called a path.[8] Figure 4.4 shows the algorithm **Initialize-Path**, which is used to convert a procedure's first demonstration into a path and to convert clarification demonstrations into paths.

Demonstration:
- **Type**: add-step
- **Prefix**: prefix1
- **Previous-step**: begin-proc1
- **Steps**:
  - turn-1 $\rightarrow$ move-2nd-2 $\rightarrow$ turn-3

Prefix: prefix1
- **Configuration**: config1
- **Additional-actions**: none

Step turn-1:
- **Operator**: turn
- **Action-example**: example1

Step move-2nd-2:
- **Operator**: move-2nd
- **Action-example**: example2

Step turn-3:
- **Operator**: turn
- **Action-example**: example3

Figure 4.3: First Demonstration

[8]The data structures for both paths and demonstrations are defined in Section 4.3.

```
procedure **Initialize-Path**

Input:   demo: A demonstration
         pname: The procedure's name
Result:  pth: A new, initialized path.

1. If the demonstration is of type add-step then the path will be
   used to create a plan for the procedure.

      generates-plan(pth) ← yes

   Otherwise, the path will not be used to create a plan

      generates-plan(pth) ← no

2. Copy the information necessary to restore the demo's initial state

      prefix(pth) ← prefix(demo)

3. Copy the demonstration's steps.

      steps(pth) ← steps(demo)

4. Use the procedure name pname to create step names for
   the procedure's beginning (begin-pname) and end (end-pname).

5. Adjust the path's steps so that the step representing the
   beginning of the procedure is the first step and the step
   representing the end of the procedure is the last step.
```

Figure 4.4: Initializing a Path

Generates-Plan: Yes
Prefix: prefix1
Steps:
begin-proc1 → turn-1 → move-2nd-2 → turn-3 → end-proc1

Figure 4.5: The Initial Path

The path created from the demonstration in Figure 4.3 is shown in Figure 4.5. In Figure 4.5, the step begin-proc1 represents the start of the procedure, and the step end-proc1 represents the end of the procedure.

4.6.6 A Second Demonstration

So far Diligent has recorded information about the new procedure in a path, but there may be problems with this information. To correct any problems, the instructor needs to be able to modify a path. Diligent allows instructors to modify paths by performing additional demonstrations that add steps to the path.[9]

Some reasons for adding additional steps to a procedure include

- The instructor wants to elaborate the procedure by adding more steps.
- The instructor wants to correct an error or a problem with the existing steps.

The use of additional demonstrations is limited by Diligent's assumption that the steps in a path represent a linear sequence of actions that transform the path's initial state into its final state. This assumption supports plans where unnecessary steps can be skipped at run-time, but the assumption doesn't support plans containing alternative steps for different initial states. Nevertheless, the assumption is used because it simplifies the derivation of the plan's step relationships and because the assumption reflects Diligent's assumptions about demonstrations (Section 4.4).

To illustrate the algorithms for combining demonstrations, we will add a step to the running example. (Of course, such a simple procedure should only need one demonstration.)

To augment the procedure, the instructor could have shut additional valves. However, to simplify the procedure, the instructor will only add a single additional step. We will assume that the handle (handle1) that is used to shut valves should be stored in a standard location (i.e. on top of valve1). This means that the instructor will need to move the handle to valve1.

Now suppose that the instructor starts a new demonstration and indicates that it is an add-step demonstration.

[9]Diligent also allows an instructor to delete steps, but deleting steps is an editing feature that we will not discuss.

4.6.6.1 Setting Up the Demonstration's Initial State

The first problem is specifying the new demonstration's initial state. One approach would be to restore the path's initial state and then have the instructor perform steps that put the environment in state where the new step could be performed. However, this approach has a few problems. The instructor has to duplicate steps from the previous demonstration. This not only takes time but is also a potential source of errors. Instead, Diligent takes a different approach. Diligent has the instructor specify an existing step that is before the new demonstration. Diligent then performs the procedure through the specified step.

Now suppose that the instructor indicates that the new demonstration should start after the last step (turn-3) in the procedure's path. This means that the new demonstration will start in the previous demonstration's final state.

To do this, Diligent uses **Replay-Prefix** (Figure 4.6) and the path's prefix (prefix1 in Figure 4.3) to restore the path's initial state.

procedure **Replay-Prefix**

Input: *pre*: A prefix
Result: Resets the state of the environment

1. Use configuration(*pre*) and **Restore-Environment-State** (Section 3.1.3) to restore the environment to a known state.

2. Now make additional changes using the sequence of actions in additional-actions(*pre*). This is done by invoking **Perform-Action** (Section 3.1.3).

Figure 4.6: Using a Prefix

After restoring the path's initial state, Diligent performs all three of the previous demonstration's steps (i.e. turn-1, move-2nd-2 and turn-3). This results in the new demonstration having the prefix shown in Figure 4.7. The prefix's additional-actions represent the action-ids needed to perform the path's existing steps.

4.6.6.2 Performing The Demonstration

The instructor then performs the demonstration by doing the following. The instructor moves the handle from valve2 to valve1 by selecting valve1 with the mouse. Since no

Prefix: prefix2
  Configuration: config1
  Additional-actions: turn handle1 $\rightarrow$ move valve2 $\rightarrow$ turn handle1

Figure 4.7: The Second Demonstration's Prefix

matching operator is found, Diligent creates a new operator. The instructor calls the operator "move-1st" and approves the default description "move to the first stage valve." The new step is called move-1st-4. At this point, the instructor ends the demonstration.

4.6.6.3 Processing the Demonstration

Demonstration:
  Type: add-step
  Prefix: prefix2
  Previous-step: turn-3
  Steps: move-1st-4

Step move-1st-4
  Operator: move-1st
  Action-example: example4
    Action-id: move valve1
    Pre-state:
      (valve1 shut)(valve2 shut)(HandleOn valve2)(AlarmLight1 off)
      (CdmStatus normal)
    Delta-state:
      (HandleOn valve1)

Figure 4.8: The Second Demonstration

As Diligent observes the demonstration, it records the data shown in Figure 4.8.

Diligent then uses this data to insert the demonstration's step into the path. The algorithm for doing this is shown in Figure 4.9. In Figure 4.9, line 1 is used when a demonstration adds steps to the start of the procedure. In this case, the path's prefix is replaced by the demonstration's prefix because the new steps might be dependent on the demonstration's prefix.

The demonstration could have a different prefix than the path because the instructor could have added additional actions to the path's prefix. For example, the instructor might

```
procedure **Add-Demo-To-Path**

Input:    demo: An add-step demonstration
          pth: A path that is used to generate a plan.
Result:   The demonstration is incorporated into the path.

1. If the demonstration's previous-step is the step representing the
   procedure's initial state (e.g. begin-proc1), then the demonstration
   adds steps to the start of the procedure. In this case, replace the
   path's initial state (i.e. prefix) with the demonstration's.

      $\text{prefix}(pth) \leftarrow \text{prefix}(demo)$

2. Insert the demonstration's sequence of steps $x_1 \ldots x_d$
   into the path's sequence of steps $s_1 \ldots s_n$.

   If the demonstration's previous-step is $s_j$ then
      $\text{steps}(pth) \leftarrow s_1 \ldots s_j x_1 \ldots x_d s_{j+1} \ldots s_n$
```

Figure 4.9: Adding a Demonstration to a Path

want students to perform an additional step. He could do this by modifying the path's prefix so that an additional step was required to successfully perform the procedure.

When using the example demonstration, line 1 in Figure 4.9 is skipped because the demonstration adds steps to the end of the procedure. The updated path is shown in Figure 4.10.

Generates-Plan: Yes
Prefix: prefix1
Steps:
begin-proc1 → turn-1 → move-2nd-2 → turn-3 →
move-1st-4 → end-proc1

Figure 4.10: Updated Path

4.6.7 Generating a Plan

We now have a path that defines the procedure's steps, but a path is not usable as a procedure. A path only contains a linear sequence of steps and does indicate how the steps are related to each other.

As mentioned in Section 4.3.5, a procedure consists a set of paths and a plan (Section 3.2.2.1). In the following sections, we will discuss how the data in a path is transformed into a plan.[10]

4.6.7.1 Guessing the Procedure's Goals

The procedures learned by Diligent attempt to put the environment into a given state. When the state is reached, the procedure is finished. This state is called the *goal state* and is defined by a set of goal conditions that need to be satisfied. A *goal condition*, like any other condition, is specified by an attribute and its value. Procedures that terminate when the environment is put into a given state are said to have *goals of attainment* [Wel94].

Besides attributes that are present in the environment, goal conditions can also include conditions that represent the values of mental attributes. A *mental attribute* is internal to Diligent and contains information that Diligent has collected during the procedure. For example, a mental attribute might record that the instructor explicitly checked whether a light was illuminated.[11]

Since Diligent finishes a procedure when all the procedure's goals have been attained, mental attributes allow Diligent to perform the steps in a path even if the steps cause no net change in the environment's state. Thus, other systems that only use goals of attainment but do not have mental attributes (e.g. Instructo-Soar [HL95]) cannot learn this type of procedure.

Diligent attempts to aid the instructor by identifying likely goal conditions. Diligent can do this because it is learning goals of attainment and because the action-examples associated with each step indicate how the environment changed during that step. Diligent hypothesizes that attributes that changed value during one of the procedure's steps are involved in a goal condition.

This heuristic technique ignores attributes whose values are constant during a procedure. Although the values of these attributes could be goal conditions, there is no evidence to indicate that they are goal conditions.

The technique for identifying goals was borrowed from Instructo-Soar [HL95]. However, Instructo-Soar only looks for attributes with different values in the initial and goal states. In contrast, Diligent looks for attributes that change value during at least one step. This

[10]As mentioned before, all paths but one represent clarification demonstrations. Clarification demonstrations provide additional data for machine learning without adding steps to the procedure's plan.

[11]Section 4.7.4 discusses sensing actions and mental attributes in more detail.

difference allows Diligent to identify attributes whose value is the same in the initial and goal states but changes during the procedure.

Diligent's technique has some advantages over Instructo-Soar's. Diligent can identify a larger set of candidate goal conditions. Furthermore, if an instructor makes an effort to undo state changes from earlier in the path, then the values of the attributes involved might be important. Consider an example from a machine maintenance domain. When diagnosing a problem, a device might be kept in standard state. During a diagnostic procedure, a human might perform actions to gather information about the state of the device before returning the device to the standard state. In this case, the conditions involved in the standard state are important even if they are the same in the initial and goal states.

Potential goal conditions are calculated from a path using **Derive-Path-Goals** (Figure 4.11). All the steps in our running example's path are primitive (line 4). The goal conditions derived from our path are shown in Figure 4.12. The condition (HandleOn valve1) is a goal condition even though the value of attribute HandleOn is the same at the beginning and end of the path.

4.6.7.2 Deriving Step Relationships

Once the procedure's goals are known, Diligent can attempt to determine how each step supports establishing the procedure's goal conditions. Steps can do this by directly satisfying goal conditions or satisfying preconditions of later steps.

Diligent records the relationships between steps in what we will call step relationships. *Step relationships* consist of causal links and ordering constraints. A *causal link* indicates that a state change of an earlier step is a precondition for a later step, and an *ordering constraint* indicates the relative order for performing a pair of steps.[12]

Step relationships are updated with **Update-Step-Relationships** (Figure 4.13).

The data available for computing step relationships consists of the procedure's goal conditions and a path, which contains a linear sequence of steps. Steps contain the following information:

- An operator that is independent of the procedure.
- An action-example that indicates the environment's state before and after the step.
- Step-specific control-preconditions that may not be required by the operator.

[12] The plan representation, including causal links and ordering constraints, is discussed in Section 3.2.2.1.

```
procedure **Derive-Path-Goals**

Input:    pth: A path that is used to generate a plan.
Output:   goals: A set of goal conditions.

1. For each step stp in the path do the following. Start with the
   last step and iterate backwards through the sequence of steps

    2. If the step represents the procedure's initial or goal states then
       do nothing.

    3. If the step represents an abstract step (i.e. subprocedure), then
       add the subprocedure's goal conditions to goals if there is not any
       condition in goals with the same attribute.

       $goals \leftarrow goals \cup \{c_1 \mid c_1 \in$ subprocedure-goals$(stp) \wedge$
               $\neg\exists\ c_2 \in goals$ where attribute$(c_1)$ = attribute$(c_2)\}$

    4. If the step stp represents a primitive step

       $goals \leftarrow goals \cup$ conditions generated by $stp$ that
               involve mental attributes.

       Also add any delta-state conditions of the step's action-example
       that do not have the same attribute as a condition in goals.

       $goals \leftarrow goals \cup \{c_1 \mid$ ex = action-example$(stp) \wedge$
               $c_1 \in$ delta-state(ex) $\wedge \neg\exists\ c_2 \in goals$ where
               attribute$(c_1)$ = attribute$(c_2)\}$
```

Figure 4.11: Deriving Goals from a Path

(valve1 shut)(valve2 shut)(HandleOn valve1)

Figure 4.12: Goal Conditions Derived from Path

procedure **Update-Step-Relationships**

Input: *proc*: The procedure.
pth: The path containing the procedure's steps.
Result: A procedure with updated causal links and ordering constraints.

1. Use path *pth* and **Derive-Path-Effect-Skeleton** to create a skeleton for the path. The skeleton indicates which operator effects are associated with each step. The skeleton is an intermediate calculation.

2. Use the path's skeleton and **Derive-Causal-Links** to generate a set of candidate causal links (*cl-cand*). The procedure also creates a proof, which identifies which operator effects achieve the procedure's goals.

3. Use the proof and **Derive-Ordering-Constraints** to generate a set of candidate ordering constraints (*ord-cand*).

4. For every causal link in *cl-cand*, add an ordering constraint between the causal link's two steps to *ord-cand*.

Figure 4.13: Computing Step Relationships

- Conditions containing mental attributes (mental-conditions) that are established by the step. Mental attributes are internal to Diligent and are not part of the environment.

Unfortunately, the data associated with steps is not in a form that can easily be used. Therefore, Diligent simplifies the data representation with **Derive-Path-Effect-Skeleton** (line 1). The procedure **Derive-Path-Effect-Skeleton** combines the data for a step's operator, action-example and mental-conditions in order to identify the step's preconditions and state changes. This data structure is called a *skeleton* because it is in an unfinished state and because it provides a framework that identifies the procedure's sequence of steps, their preconditions, and their state changes.

Operator: turn
- **Action-id**: turn handle1
- **Effect**: effect1
  - **H-rep preconditions**: (valve1 open)
  - **State changes**: (valve1 shut)
- **Effect**: effect2
  - **H-rep preconditions**: (valve1 shut)(valve2 open) (HandleOn valve2)
  - **State changes**: (valve2 shut)

Operator: move-1st
- **Action-id**: move valve1
- **Effect**: effect3
  - **H-rep preconditions**: (HandleOn valve2)
  - **State changes**: (HandleOn valve1)

Operator: move-2nd
- **Action-id**: move valve2
- **Effect**: effect4
  - **H-rep preconditions**: (valve1 shut)(HandleOn valve1)
  - **State changes**: (HandleOn valve2)

Figure 4.14: The Operators

The algorithm for **Derive-Path-Effect-Skeleton** will be illustrated with our running example. During the example's two demonstrations, operators were created. These operators are shown in Figure 4.14. Operators were defined in Section 3.2.2.2, but we will briefly review them. An *operator* models how an action affects the environment. Since actions can produce different state changes in different situations, an operator models different state changes with different *conditional effects* (or *effects*). Each effect identifies a set of preconditions that must be met for the given state changes to appear. While an effect has three sets of preconditions, Diligent only uses the best guess, heuristic set of preconditions (h-rep) when creating a plan.

A problem with the operators in Figure 4.14 is that Diligent has only observed each effect performed once. Because of this lack of data, the preconditions contain some errors. For example, effect1 is missing the precondition (HandleOn valve1). Unfortunately, missing preconditions can cause missing step relationships, and unnecessary preconditions can cause unnecessary step relationships. (In Chapter 6, we will discuss how to correct preconditions by performing experiments.)

We are now ready to discuss **Derive-Path-Effect-Skeleton** (Figure 4.15). The procedure traverses the path sequentially going from path's first step to its last step. For each step, the algorithm identifies operator effects that transform the state before the step (pre-state) into the state after the step (post-state).

Notice that steps that represent an action (primitive steps) are treated differently than subprocedures (abstract steps). On line 5, Diligent simulates performing a subprocedure in order to determine which of its steps are performed when starting in the abstract step's pre-state. Given the subprocedure's steps, Diligent can compute the abstract step's preconditions. Diligent simulates the subprocedure each time the skeleton is created because the instructor may have modified the subprocedure. Another concern is that a subprocedure can have state changes that are incidental and unimportant. For this reason, lines 6 and 7 only use the subprocedure's goal conditions. By creating an effect (line 7) and then adding it to the skeleton (line 8), subsequent processing can treat a subprocedure like a primitive step.

Finally, line 12 incorporates conditions involving mental attributes. Because mental attributes are internal to Diligent, they are not stored in action-examples, which record the state of the environment.

The computation of the skeleton assumes that each action-example has the correct delta-state because the instructor demonstrated all steps correctly. That the instructor demonstrates steps correctly seems a reasonable assumption, especially if most procedures are relatively short. However, if the instructor makes a mistake and has to provide another demonstration, some step's action-example may be incorrect.[13]

Figure 4.16 shows a skeleton for our procedure using the path in Figure 4.10, the operators in Figure 4.14, and the action-examples in Figures 4.1 and 4.8. Notice that

[13]The problem of correcting a step's action-example is not addressed by Diligent.

```
procedure **Derive-Path-Effect-Skeleton**

Input:    pth: A path
Result:   skeleton: Identifies the operator effects used by each of
              the path's steps. The order of the path's steps is maintained.

1. Initialize skeleton as empty.
2. For each step stp in the path do the following
        3. If the step represents the beginning or end of the procedure,
            do nothing.
        4. If the step represents a subprocedure then
                5. Use Internally-Simulate-Subprocedure to determine
                    the subprocedure's preconditions. (Section 4.7.1)
                6. Get the subprocedure's goal conditions.
                7. Create an effect using the preconditions from 5 with the
                    state changes of 6.
                8. Associate the effect with the step in skeleton.
        9. Else the step represents an action.
                10. Identify the effects effs of the step's operator op
                    that match the delta-state of the step's action-example ex.

                    $effs \leftarrow \{ e_1 \mid e_1 \in \text{effects}(op) \wedge$
                        $\text{state-changes}(e_1) \subseteq \text{delta-state}(ex)\}$

                11. Associate effs with the step in skeleton.
                12. If stp produces conditions containing mental attributes, then
                    create a new effect that has the conditions as its state changes.
                    The new effect has no preconditions. Add the effect to skeleton.
```

Figure 4.15: Identifying a Path's Effects

Order of steps:
- turn-1 → move-2nd-2 → turn-3 → move-1st-4

Step: turn-1
- **Effect**: effect1
  - **Preconditions**: (valve1 open)
  - **State changes**: (valve1 shut)

Step: move-2nd-2
- **Effect**: effect4
  - **Preconditions**: (valve1 shut)(HandleOn valve1)
  - **State changes**: (HandleOn valve2)

Step: turn-3
- **Effect**: effect2
  - **Preconditions**: (valve1 shut)(valve2 open)(HandleOn valve2)
  - **State changes**: (valve2 shut)

Step: move-1st-4
- **Effect**: effect3
  - **Preconditions**: (HandleOn valve2)
  - **State changes**: (HandleOn valve1)

Figure 4.16: Skeleton of Procedure

steps turn-1 and turn-3 are associated with the same operator but are compatible with different effects.

Once Diligent has identified the effects used by the path's steps, it can determine which effects help achieve the goal conditions. This is important because effects can also produce irrelevant state changes.

Diligent identifies the effects that achieve the procedure's goal conditions while calculating the causal links. These useful effects are stored in a data structure called a proof. It is called a *proof* because it records how the preconditions and state changes of the path's steps transform the path's initial state into its goal state.

Diligent does this calculation with **Derive-Causal-Links** (Figure 4.17). The algorithm treats the goal conditions as preconditions of the goal state step (line 1). The algorithm iterates backwards over path's sequence of steps starting at the end of the procedure (line 2). When iterating over the steps, preconditions of later steps are used as indices into the array *dstnam*. Because the path's sequence of steps is known to achieve the goal conditions, Diligent identifies earlier steps that establish the preconditions of later steps. When a state change of an earlier step is found to establish a precondition of a later step, a causal link is created (line 5). Because the precondition has been established, it

```
procedure **Derive-Causal-Links**

Input:   skeleton: Identifies effects produced
             by the path's steps.
         proc: The procedure.
Result:  cand: Set of candidate causal links.
         proof: Similar to skeleton but only contains the effects
             that help achieve the procedure's goal conditions.

(The following uses the array dstnam that is indexed by a condition.
    Each element contains a set of steps that have the condition as a
    precondition.)

1. For each of the procedure's goal conditions gcnd, add the
    goal state step to dstnam(gcnd).
2. Iterate over each step stp in skeleton starting with the
    path's last step and working backwards to the first step.
        3. For each effect eff of stp in skeleton do the following

                4. If a condition cnd in eff's state change has an
                    element in dstnam, then eff is needed to achieve
                    the procedure's goal conditions. In this case, do the following.

                        5. Add causal links to cand for condition cnd between
                            step stp and later steps that have cnd as a precondition.
                            These later steps are identified by dstnam(cnd).
                        6. After adding the causal links, remove dstnam(cnd) in
                            order to prevent spurious causal links.
                        7. Add eff to proof for step stp.

        8. Create an effect for the stp's control-preconditions and add it to proof.
            The new effect will have preconditions but no state changes.
        9. Now add stp's preconditions to dstnam. (For each effect eff of stp in
            proof and for each precondition pcond of eff, add stp to dstnam(pcond).)
10. Any elements left in dstnam are dependent on the procedure's initial
    state. For each element of dstnam add a casual link for that condition from
    the initial state step to each of the steps listed for that condition in dstnam.
```

Figure 4.17: Computation of Causal Links

is removed from *dstnam* (line 6). The algorithm also adds effects that produce useful state changes to the proof (line 7). Line 8 adds the step's control-preconditions to the proof. Control precondition's control when the step is applicable, but may not be required by the environment. For example, a control precondition might require that a light be turned on before opening a valve. After processing a step's state changes, Diligent adds the preconditions of the step's useful effects to *dstnam* (line 9). After all the steps have been processed, any preconditions that haven't been established must rely on the initial state (line 10).

In Figure 4.17, the use of the array *dstnam* greatly reduces the run-time overhead. Each of a step's state changes is checked against one array element rather than against the preconditions of each of the path's later steps.

casual links:

a) begin-proc1	establishes	(valve1 open)	for turn-1
b) begin-proc1	establishes	(HandleOn valve1)	for move-2nd-2
c) begin-proc1	establishes	(valve2 open)	for turn-3
d) turn-1	establishes	(valve1 shut)	for move-2nd-2
e) turn-1	establishes	(valve1 shut)	for turn-3
f) turn-1	establishes	(valve1 shut)	for end-proc1
g) move-2nd-2	establishes	(HandleOn valve2)	for turn-3
h) move-2nd-2	establishes	(HandleOn valve2)	for move-1st-4
i) turn-3	establishes	(valve2 shut)	for end-proc1
j) move-1st-4	establishes	(HandleOn valve1)	for end-proc1

Figure 4.18: Causal Links

Now suppose that **Derive-Causal-Links** is used with the skeleton in Figure 4.16 and the goal conditions in Figure 4.12. Because all the operator effects in the skeleton are needed, the proof produced by the skeleton is the same as the skeleton (Figure 4.16). The resulting causal links are shown in Figure 4.18. The steps begin-proc1 and end-proc1 represent the procedure's initial and goal states, respectively. In Figure 4.18, row a) indicates that the procedure's initial state (begin-proc1) establishes the condition (valve1 open), which is a precondition for step turn-1.

Once Diligent has created a proof of the path, Diligent can compute the ordering constraints between the steps. As mentioned earlier, ordering constraints indicate the relative order for performing a pair of steps. Diligent's calculation of ordering constraints is simpler than what would be seen in partial ordered planner [Wel94] because Diligent already knows a sequence of steps that will correctly perform the procedure. For this reason, Diligent does not have to consider rearranging a procedure's steps.

Usually, there is an ordering constraint for each causal link, but more ordering constraints may be needed. Consider two steps that were demonstrated sequentially. Suppose a precondition of the first step was removed by a state change of the second step. If the first step were to be performed earlier in the procedure, the second step would not interfere with the first step, but if the second step were to be performed immediately in front of the first step, the first step's precondition would not be satisfied, and the first step would not cause a necessary state change.

In this situation, Diligent adds an ordering constraint to prevent the state change of the later step from removing a precondition of the earlier step. The technique of adding an ordering constraint to a procedure so that a later step doesn't remove a precondition of an earlier step is called *promotion* [Wel94].

In Figure 4.19, **Derive-Ordering-Constraints** only uses promotion to derive ordering constraints. The array *clobberstp* improves run-time efficiency because a precondition is only checked against steps that change the precondition's attribute rather than against all later steps. The ordering constraints associated with causal links are calculated in **Update-Step-Relationships** (line 8 in Figure 4.13).

The ordering constraints for our running example are shown in Figure 4.20. Any ordering constraints involving the procedure's initial state and goal state are ignored because, by definition, the initial state is before all steps and the goal state is after all steps. The ordering constraints created with the procedure **Derive-Ordering-Constraints** are listed as being created by promotion.

At this point, the instructor is finished with procedure proc1. The plan is shown in Figure 4.21.

4.7 Creating a Hierarchical Procedure

The techniques that we've looked at so far have problems scaling to larger procedures. We need to be able to divide procedures into modular tasks, and we should be able to reuse existing procedures.

Diligent addresses this issue with hierarchical procedures. A *hierarchical* procedure uses another procedure as one of its steps. A procedure used as a step in another procedure is called a *subprocedure*, and the procedure containing the subprocedure is called the *parent*

```
procedure **Derive-Ordering-Constraints**

Input:    proof: Contains the effects needed by each
              step to achieve the procedure's goals.
Output:   cand: Set of candidate ordering constraints.

(The following uses the array clobberstp that is indexed by an
    attribute. Each element contains a set of steps that change the
    attribute's value. The array is used to reduce searching.)

1. Iterate over each step stp in proof starting with the
    path's last step and working backwards to the first step.

        (Check stp against the steps later in the path.)

        2. For each precondition pcond of stp in proof do the following.

                3. If the pcond is not equal to a condition for the
                    same attribute in a later step's (stp2's) state changes then
                    add an ordering constraint between the two steps to cand.

                $cand \leftarrow cand \cup \{ ord_i \mid$ where $ord_i$ is an ordering constraint
                    between steps $stp$ and $stp2 \wedge eff$ is an effect of $stp$ in $proof \wedge$
                    $pcond \in \text{precondition}(eff) \wedge attr = \text{attribute}(pcond) \wedge$
                    $stp2 \in \text{clobberstp}(attr) \wedge cond \in \text{state-change}(stp2) \wedge$
                    $attr = \text{attribute}(cond) \wedge \text{value}(cond) \neq \text{value}(pcond)\}$

        (Prepare to check stp against steps earlier in the path.)

        4. For each state change condition of stp in proof, add stp to
            the set of steps in clobberstp using the condition's
            attribute as an index.
```

Figure 4.19: Computation of Additional Ordering Constraints

Ordering constraints:

Created by promotion:		
turn-3	before	move-1st-4
Created by causal links:		
turn-1	before	move-2nd-2
turn-1	before	turn-3
move-2nd-2	before	turn-3
move-2nd-2	before	move-1st-4

Figure 4.20: Ordering Constraints

procedure. A step representing a subprocedure is called an *abstract* step, while other steps are called *primitive* steps.

4.7.1 Internally Simulating A Subprocedure

When a procedure is created, its steps reflect the initial state of its path. However, when a procedure is used as a subprocedure, it may have a different initial state. This means that some of the procedure's steps may no longer be needed. To overcome this problem, Diligent can internally simulate performing a subprocedure.

Diligent also internally simulates the performance of subprocedures for other purposes. Diligent simulates a subprocedure when computing step relationships in order to determine the preconditions of the subprocedure's abstract step. Diligent also simulates a subprocedure when figuring out which subprocedure steps to perform during one of its experiments (Chapter 6).

A subprocedure has the same semantics as a STRIPS macro-operator [RN95], and the criteria used by Diligent for determining when to perform a step was developed by Jeff Rickel for the STEVE tutor [RJ99].[14] STEVE examines the current state and determines which steps are needed to achieve the goal conditions.

However, unlike STEVE, Diligent cannot assume that a step's preconditions are correct. If a primitive step's operator is not very refined, then the step could have unnecessary or missing preconditions. A missing precondition could cause Diligent to skip the step that establishes the precondition, and an unnecessary precondition could prevent a necessary step from being performed because the precondition is never satisfied.

[14]Diligent and STEVE were developed as part of the same project [JRSM98].

Steps:

begin-proc, turn-1, move-2nd-2, turn-3, move-1st-4, end-proc1

Goal conditions:

(valve1 shut)(valve2 shut)(HandleOn valve1)

Causal links:

begin-proc1	establishes (valve1 open)	for turn-1
begin-proc1	establishes (HandleOn valve1)	for move-2nd-2
begin-proc1	establishes (valve2 open)	for turn-3
turn-1	establishes (valve1 shut)	for move-2nd-2
turn-1	establishes (valve1 shut)	for turn-3
turn-1	establishes (valve1 shut)	for end-proc1
move-2nd-2	establishes (HandleOn valve2)	for turn-3
move-2nd-2	establishes (HandleOn valve2)	for move-1st-4
turn-3	establishes (valve2 shut)	for end-proc1
move-1st-4	establishes (HandleOn valve1)	for end-proc1

Ordering constraints:

turn-1	before	move-2nd-2
turn-1	before	turn-3
move-2nd-2	before	turn-3
move-2nd-2	before	move-1st-4
turn-3	before	move-1st-4

Figure 4.21: The Plan for Procedure proc1

```
procedure **Internally-Simulate-Subprocedure**

Input:    pre-state: The subprocedure's initial state.
          proc: The subprocedure.
Result:   used-steps: Sequence of steps that achieves proc's goals.
          pcond: The preconditions in pre-state.

(In the following, a needed precondition, goal condition or step
  is called relevant. A step is only assumed to have
  a precondition when the step is enabled by a causal link
  that establishes that precondition.)

1. If proc does not yet have any causal links, add
   all of proc's steps to used-steps, set
   pcond to be empty, and return.

2. Compute the steps that are needed to achieve proc's goal
   conditions. This is done by iterating backwards over the
   procedure from the goal conditions to the start of the procedure.

     i) All goal conditions are relevant.

     ii) A step is relevant if it establishes an unsatisfied goal
         condition or an unsatisfied but relevant precondition.

     iii) The conditions of all causal links that enable a relevant step
          are relevant preconditions of that step.

     iv) Relevant preconditions are satisfied when the causal link
     associated with the precondition is established by either
     another step or the subprocedure's pre-state.

3. Add all relevant steps to used-steps. While adding steps
   maintain the same step order as the procedure.

4. If a relevant precondition or goal condition is not established
   by a relevant step and the condition is true in the subprocedure's
   pre-state, then add the condition to the subprocedure's
   pcond.
```

Figure 4.22: Simulating a Subprocedure

The algorithm to simulate performing a subprocedure is shown in Figure 4.22. The calculation is called “simulation” rather than “planning” because it uses a linear sequence of steps (i.e. a path) and determines which steps will achieve the subprocedure’s goal conditions.

Line 1 deals with the situation when a subprocedure doesn’t yet have any step relationships. One solution is to force the instructor to define goal conditions and step relationships. However, this approach is intrusive, and the goal conditions and step relationships may not yet be necessary. To keep the interaction with the instructor simple, Diligent assumes all the path’s steps should be performed if a procedure has no causal links. In this case, because preconditions depend on causal links, no preconditions can be found.

Line 2 determines which steps are needed to achieve the procedure’s goal conditions. This calculation is similar to the calculation used by STEVE [RJ99].

Line 3 just gathers that the steps that Step 2 identified as relevant (i.e. need to be performed).

Line 4 identifies the preconditions of the subprocedure, but is different than what STEVE would do. STEVE would include all preconditions that were marked as relevant, while Diligent only includes preconditions that are satisfied in the subprocedure’s initial state. To see why Diligent took this approach, suppose that an existing procedure is reused as a subprocedure. In this case, some of the subprocedure’s preconditions might be unsatisfied. Since the preconditions are unsatisfied in the subprocedure’s initial state, they cannot be used as preconditions of the subprocedure. (If the subprocedure can achieve its goals from this initial state, these unsatisfied preconditions were unnecessary.)

The major problem with simulating a subprocedure is compensating for possible errors in the preconditions of steps, especially the initial preconditions created with heuristics. Diligent handles this problem by utilizing the fact that its heuristics for learning preconditions favor creating unnecessary preconditions over skipping potentially necessary ones.[15] For this reason, it is sometimes reasonable for Diligent to ignore unsatisfied preconditions (line 4).

Another issue is dealing with abstract steps embedded within a subprocedure. In this case, Diligent assumes that the causal links involving the abstract steps are reasonable. This allows Diligent to treat abstract steps the same as primitive steps and reduces the overhead of simulating the abstract steps inside a subprocedure.

[15]The reasons that the heuristics favor unnecessary preconditions will be discussed in Chapter 5.

From this discussion it may seem that the reuse of subprocedures is undesirable. However, reusing a subprocedure saves time, and performing a subprocedure under different initial states helps refine the preconditions of the subprocedure's steps.

A problem that Diligent does not address is when the internal simulation does not correctly identify the steps needed to achieve the subprocedure's goal conditions. Ideally, Diligent would notice this, notify the instructor, and interact with him in order to fix the problem. This type of dialog is supported by Instructo-Soar [HL95].[16]

4.7.2 Continuing the Running Example

Now let us return to our running example. We will create a hierarchical procedure that contains three steps, two of which are abstract. We will look at the two ways of inserting subprocedures into a parent procedure.

- An existing procedure can be inserted as a subprocedure. This can save an instructor time and effort.
- A new procedure can be defined as a subprocedure inside a demonstration of the parent procedure. This can be a convenient way of authoring a subprocedure in the desired initial state.

Suppose the instructor now authors a hierarchical procedure that shuts some valves and checks whether a light works. The instructor will use the same initial state as our first procedure. The instructor calls the procedure "top-level" and gives it the description "perform a hierarchical procedure." The instructor then demonstrates the new procedure.

1. The Instructor turns the handle and shuts valve1. This step is called turn-5.
2. The second step reuses our first procedure proc1 (Figure 4.21). The instructor uses proc1 by selecting it from a menu of potential subprocedures. This step is called proc1-6.
3. The third step is a new procedure that checks whether an alarm light is working. The new procedure is defined inside the demonstration of its parent procedure (top-level). The instructor calls the new procedure "proc2" and gives it the description "check the alarm light." The instructor finishes demonstrating and defining proc2 before continuing the demonstration of procedure top-level. This step is called proc2-7.

[16]Extensions to support unexpected behavior in subprocedures are discussed in Chapter 8.

Step turn-5 is a redundant step that performs the work of the first step (turn-1) in procedure proc1 (step proc1-6). Step turn-5 is used to show why subprocedures need to be internally simulated. Even though step turn-5 is a primitive step, it is meant to illustrate the situation where the state changes of one subprocedure interact with the preconditions of a later subprocedure.

Because step turn-5 performs the first step of subprocedure proc1, Diligent needs to simulate proc1 so that it can determine which steps to perform and identify the preconditions of step proc1-6. Diligent simulates the subprocedure using the post-state of step turn-5 and **Internally-Simulate-Subprocedure** (Figure 4.22). As expected, Diligent determines that the subprocedure's step turn-1 is unnecessary because the condition (valve1 shut) has already been established. After doing the simulation, Diligent performs the abstract step proc1-6 by performing the steps shown in Figure 4.23.

In Figure 4.23, the abstract step proc1-6 is also associated with an action-example. Diligent creates an action-example for an abstract step by recording the state before and after performing the step.

Steps to perform:
move-2nd-2 → turn-3 → move-1st-4

Preconditions:
(valve1 shut)(valve2 open)(HandleOn valve1)

Action-example:
Pre-State:
(valve1 shut)(valve2 open)(HandleOn valve1)(AlarmLight1 off)
(CdmStatus normal)
Delta-State:
(valve2 shut)

Figure 4.23: Results from Simulating Step proc1-6

4.7.3 A Nested Procedure Definition

After Diligent has performed abstract step proc1-6, the instructor defines a new procedure inside the current demonstration. The concept of nested procedure definitions has been borrowed from Instructo-Soar [HL95].

To construct the new subprocedure's prefix, the prefix of the parent procedure has appended to it every action necessary to reach the subprocedure's initial state.[17] When constructing the prefix, abstract steps are represented by their primitive steps, and primitive steps are represented by their associated actions. Figure 4.24 shows the prefix for the new subprocedure (proc2).

Prefix: prefix3
 Configuration: config1
 Additional-actions:
  turn handle1 → move valve2 → turn handle1 → move valve1

Figure 4.24: The Subprocedure's Prefix

4.7.4 Sensing Actions

The new procedure (proc2) checks whether a light is working. The procedure illustrates the use of a step that performs an information gathering (or sensing) action [AIS88, RN95]. A *sensing* action gathers information about the environment without changing it. This raises three immediate issues:

1. What are the step's preconditions? The environment may place no restrictions on when the sensing action can be performed.
2. How does Diligent indicate that the step has been performed? Because the sensing action does not change the environment, what is to prevent the step from being repeated indefinitely?
3. Since the step doesn't change the state, what is to prevent Diligent from just skipping the step?

Diligent addresses these issues by creating preconditions that control when the step is performed and creating internally maintained mental attributes. A *mental* attribute is an attribute that is maintained inside Diligent and is not present in the environment. A sensing action creates a condition involving a new mental attribute, and the condition is incorporated into the procedure's goal conditions [RJ99]. Adding the goal condition ensures that the sensing action is performed once.

[17] Instructo-Soar does not use prefixes.

To control when the sensing action is performed, Diligent uses heuristics to create provisional preconditions for the sensing action's step. While creating the preconditions, Diligent focuses on the current demonstration of the current procedure. Diligent assumes that attributes that change value are likely to be important. Since earlier steps are likely to establish preconditions for later steps, the state changes caused by earlier steps are likely preconditions. The preconditions of sensing actions are calculated with **Compute-Changes-in-Demo** (Figure 4.25), which is invoked during a demonstration when a sensing action is performed.

```
procedure Compute-Changes-in-Demo

Input:    demo: A demonstration.
          cur-state: The environment's current state.
Output:   chgs: A set of state changes.

(The set attrs contains the names of attributes that change value
during the demonstration.)

1. For each step in demo, add the attribute of every condition in the
   delta-state of the step's action-example to attrs.
2. For each attribute that changed value (i.e. in attrs),
   add the attribute's condition in the current state (cur-state)
   to chgs.
```

Figure 4.25: Computing State Changes Caused by Earlier Steps

In **Compute-Changes-in-Demo**, the action-examples of abstract steps are treated the same as the action-examples of primitive steps. This means that attributes that change value in the subprocedure but have the same initial and final value are ignored. This approach simplifies processing and doesn't require a subprocedure's goal conditions or causal links to be defined. Besides, the algorithm for computing state changes only provides heuristic preconditions.

Because a sensing action might be performed at any time and because a procedure may contain several sensing actions, we are using the convention that a sensing action performed by a human student will not be recognized unless all of its preconditions are satisfied. Otherwise, a sensing action might not be performed in the proper situation, which means the sensing action would not be performed properly.

Diligent only identifies preconditions for a sensing action when the sensing action is demonstrated in an add-step demonstration. Afterwards, the preconditions only change when the instructor edits them.

In a future system, machine learning techniques could be used to refine a sensing action's preconditions if a sensing action could be demonstrated multiple times. The system might then look for commonality between the demonstrations.

However, beyond multiple demonstrations, it is unclear how to use machine learning techniques with sensing actions because they don't affect the state of the environment. Perhaps, it might be possible to use the placement and type of sensing action to make inferences about other aspects of a procedure

While this approach for identifying sensing action preconditions worked on the procedures that we looked at, it became clear during Diligent's evaluation (which did not use sensing actions) that the approach would have been more robust if it had also looked at attributes that changed value after the sensing action. Using the state changes of earlier steps places the sensing action after the earlier steps, and using attributes that change value later in the procedure would have placed the sensing action in front of later steps.

Consider the following example of why both sources of preconditions are important. Suppose a procedure involves pressing the reset button, checking if a light is illuminated, and turning off the system by pressing the power button. In this case, checking the light has no value if it is checked before the reset button has been pressed or after the power button has been pressed. By using state changes of steps both before and after the sensing action, the sensing action could have been positioned so that it was correctly performed as the second step.

4.7.5 Demonstrating the Nested Procedure

Now let us look at the demonstration of the procedure (proc2) containing the sensing action. During the demonstration, the instructor performs the following steps.

1. The instructor presses the function-test button, which causes the alarm light to turn on. The instructor calls the operator "press-test" and approves the default description "press the system test button." The step is called press-test-8.

2. The instructor performs a sensing action on the light by selecting the light with the mouse. The instructor calls the operator "check-light" and approves the default description "check the alarm light." This step is called check-light-9.

3. The instructor turns off the light by pressing the reset button. The instructor calls the operator "press-reset" and approves the default description "press the system reset button." This step is called press-reset-10.

Figure 4.26 shows information for proc2's demonstration. The conditions found by **Compute-Changes-in-Demo** are listed as the control preconditions of step check-light-9. In contrast to the preconditions of an effect, which are required by the environment to produce the effect's state changes, *control preconditions* are specific to a step and need to be true before the step is performed. For this reason, control preconditions are associated with the step rather than with the operator.

The mental attribute (AlarmLight1-result) created by step check-light-9 is added to the step's mental-conditions because Diligent associates each mental attribute with a distinct step. The value of the mental attribute is not considered important (i.e. <any value>) because none of the procedures used with Diligent could utilize the mental attribute's value. A more sophisticated use of mental attributes and sensing actions will be discussed when we talk about potential extensions (Section 8.4.2.3).

At this point, the instructor derives the procedure's goal conditions and step relationships. The plan for proc2 is shown in Figure 4.27.

4.8 The Completed Procedure

After the instructor has finished subprocedure proc2, he finishes demonstrating its parent procedure (top-level). The plan for top-level is shown in Figure 4.28. The plans for the abstract steps proc1-6 and proc2-7 have already been shown in figures 4.21 and 4.27, respectively.

One thing to note about top-level's plan is that subprocedures are treated as black boxes that achieve their goal conditions. This is done because subprocedures do not terminate until their goal conditions are satisfied. Furthermore, a subprocedure's plan supports some ability to adjust to different initial states. Moreover, treating subprocedures as black boxes simplifies processing on hierarchical procedures (e.g. computing step relationships).

Treating subprocedures as black boxes affects top-level's plan in several ways. One way is using subprocedure proc1's goal conditions for the state changes of its step when computing top-level's goal conditions (line 3 in Figure 4.11). That is why (HandleOn valve1) is a goal condition of top-level even though the condition is true in top-level's initial and goal states. Another way that subprocedures are used as black boxes is when the preconditions and goal conditions of a subprocedure (proc1) are used to create an effect

Demonstration:
- **Type**: add-step
- **Prefix**: prefix3
- **Previous-step**: begin-proc2
- **Steps**: press-test-8 → check-light-9 → press-reset-10

Step: press-test-8
- **Action-example**:
  - **Pre-state**:
    - (valve1 shut)(valve2 shut)(HandleOn valve1)(AlarmLight1 off) (CdmStatus normal)
  - **Delta-state**:
    - (AlarmLight1 on)(CdmStatus test)

Step: check-light-9
- **Action-example**:
  - **Pre-state**:
    - (valve1 shut)(valve2 shut)(HandleOn valve1)(AlarmLight1 on) (CdmStatus test)
  - **Delta-state**:
    - <*empty*>
- **Control-preconditions**:
  - (AlarmLight1 on)(CdmStatus test)
- **Mental-conditions**:
  - (AlarmLight1-result <*any value*>)
- **Operator**: check-light
  - **Effect**:
    - **Preconditions**: <*empty*>
    - **State changes**: <*empty*>

Step: press-reset-10
- **Action-example**:
  - **Pre-state**:
    - (valve1 shut)(valve2 shut)(HandleOn valve1)(AlarmLight1 on) (CdmStatus test)
  - **Delta-state**:
    - (AlarmLight1 off)(CdmStatus normal)

Figure 4.26: Subprocedure Demonstration

Steps:

begin-proc2, press-test-8, check-light-9, press-reset-10, end-proc2

Goal conditions:

(AlarmLight1 off)(CdmStatus normal) (AlarmLight1-result <*any value*>)

Causal links:

begin-proc2	establishes	(AlarmLight1 off)	for press-test-8
begin-proc2	establishes	(CdmStatus normal)	for press-test-8
press-test-8	establishes	(AlarmLight1 on)	for check-light-9
press-test-8	establishes	(CdmStatus test)	for check-light-9
press-test-8	establishes	(AlarmLight1 on)	for press-reset-10
press-test-8	establishes	(CdmStatus test)	for press-reset-10
check-light-9	establishes	(AlarmLight1-result <*any value*>)	for end-proc2
press-reset-10	establishes	(AlarmLight1 off)	for end-proc2
press-reset-10	establishes	(CdmStatus normal)	for end-proc2

Ordering constraints:

press-test-8	before	check-light-9
press-test-8	before	press-reset-10
check-light-9	before	press-reset-10

Figure 4.27: The Plan for Subprocedure proc2

when computing the path's skeleton (line 7 in Figure 4.15). This is why step proc1-6 rather than step turn-5 establishes the goal condition (valve1 shut) with causal link g).

Steps:
begin-top-level, turn-5, proc1-6, proc2-7, end-top-level

Goal conditions:
(valve1 shut)(valve2 shut)(HandleOn valve1)(AlarmLight1 off)
(CdmStatus normal)(AlarmLight1-result <*any value*>)

causal links:

a) begin-top-level	establishes	(valve1 open)	for turn-5
b) begin-top-level	establishes	(valve2 open)	for proc1-6
c) begin-top-level	establishes	(HandleOn valve1)	for proc1-6
d) begin-top-level	establishes	(AlarmLight1 off)	for proc2-7
e) begin-top-level	establishes	(CdmStatus normal)	for proc2-7
f) turn-5	establishes	(valve1 shut)	for proc1-6
g) proc1-6	establishes	(valve1 shut)	for end-top-level
h) proc1-6	establishes	(valve2 shut)	for end-top-level
i) proc1-6	establishes	(HandleOn valve1)	for end-top-level
j) proc2-7	establishes	(AlarmLight1 off)	for end-top-level
k) proc2-7	establishes	(CdmStatus test)	for end-top-level
l) proc2-7	establishes	(AlarmLight1-result <*any value*>)	for end-top-level

ordering constraints:

turn-5	before	proc1-6

Figure 4.28: The Top Level Procedure

4.8.1 Information Provided by the Instructor

To summarize the previous sections, when an instructor creates a procedure, he needs to provide demonstrations and names for procedures and operators. He must also provide English descriptions that can be used to describe procedures to human students. Descriptions of procedures are entered entirely by the instructor, but for other types of descriptions, Diligent can generate a default description. Of course, default descriptions still need to be approved (and possibly modified) by the instructor.

4.8.1.1 Generating default descriptions

Diligent provides default descriptions for operators, steps, causal links and goal conditions. These descriptions exploit Diligent's ability to query the environment for English descriptions of action-types, objects and attributes (Section 3.1.3). Diligent uses the information returned by the environment to fill in templates.

- *causal links.* The template for a causal link is "the <*attribute name*> to be <*value*>." In the template, <*attribute name*> and <*value*> represent the description of the attribute and the attribute's value, respectively. The template does not start with a complete sentence so that the tutor has flexibility in how it starts sentences. For example, the tutor might say, "Now we want the 'first valve' to be 'open'."
- *Goal conditions.* Goal conditions are represented by causal links that establish conditions for the plan's goal state step.
- *Operators.* The template is "<*type of action*> the <*object*>." For example, the tutor could use the template to say "We will now 'toggle' the 'first valve'." Of course, additional templates would be needed if operators modeled actions that involved multiple objects.
- *Steps.* By default, steps use their operator's description.

The templates are simple, but they provide the instructor with a great deal of help. They correctly identify the objects and attributes involved. Because they usually produce reasonable descriptions, they save the instructor a great deal of typing. Reducing typing not only saves time but also prevents errors.

4.9 Complexity

Because Diligent is an interactive system, its algorithms should have reasonable run-time efficiency. In this section, we will discuss the run-time complexity of simulating subprocedures and deriving step relationships.

These calculations involve identifying connections between steps, and the algorithms center on the processing of individual steps. For this reason, we will consider the processing of a step as the basic operation.

We will assume that each step has the maximum number of preconditions and these result in the maximum number of causal links and ordering constraints. For this reason, we will consider the processing on each step as approximately the same.

We will also ignore the access times of associative arrays. An associative array is indexed by a symbolic value (e.g. "blue") and can be implemented as a hash table. The worst case time for accessing an element of an associative array is linear in the number of elements in the array.

Let n = the number of steps in the current procedure without considering the steps inside subprocedures.
m = the maximum number of steps in a subprocedure without considering the steps inside a subprocedure's subprocedures. (m = n)
s = the number of subprocedures in the current procedure.
p = the maximum number of preconditions or state changes for a step.

In Diligent's algorithms, the causal links and ordering constraints are derived from the preconditions of steps. The calculations revolve around a step's preconditions rather than around causal links or ordering constraints. Thus, in the following algorithms, we will expect to process O(p) preconditions every time we process a step.

When we discuss the steps in a procedure, we mean the steps in the immediate procedure. By *immediate* procedure, we mean only the primitive and abstract steps in a procedure and not the steps inside the subprocedures associated with abstract steps.

First, we will look at simulating a subprocedure (Figure 4.22). The algorithm uses the subprocedure's causal links. This results in abstract steps inside the subprocedure being treated exactly like other steps. Determining the relevant preconditions and steps (line 2) involves visiting each of the m steps once. Later, the m steps are visited once again to identify and store the relevant steps (line 3). Because there are O(p) preconditions, we expect to process O(p) causal links for every step. Thus, the run-time complexity is O(pm).

Next, we will look at deriving step relationships (Figure 4.13). The majority of the time is spent in the algorithms that compute the path skeleton, the causal links and the ordering constraints. Each algorithm computes intermediate results that are used by the next algorithm.

The first algorithm (4.15) creates a skeleton of a path. The skeleton identifies which operator effects were used in the path. If a procedure does not contain any subprocedures, then each step is visited once (lines 9-12) and O(p) preconditions and state changes are processed. Thus, the complexity is O(pn). However, any subprocedures will need to be simulated (lines 4-8). If there are at most s subprocedures with a length of at most m, then the run-time complexity is $O(pn + spm)$.

The second algorithm (Figure 4.17) takes the skeleton and computes causal links. Each step is processed once and associative arrays are used to hold intermediate results. Because O(p) preconditions are considered, the complexity is $O(pn)$.

The third algorithm (Figure 4.19) computes ordering constraints. The algorithm looks at preconditions of steps and compares them to state changes of later steps (lines 2-3). In the worst case, every step would change the same attribute. This would result in a run-time complexity of $O(pn^2)$. However, the algorithm uses an associative array to record which attributes are changed by which steps (line 4). This reduces the expected number of comparisons. In practice, the algorithm has been very fast.

Combining the complexity for various algorithms results in a complexity $O(pn^2 + spm)$. The algorithms have been used on procedures as long 10 to 12 steps, and none of the algorithms have been observed to take more than a few seconds.

Diligent gains efficiency from its focus on the immediate procedure. The cost of simulating subprocedures is limited because Diligent uses the causal links inside subprocedures. Once an abstract step's subprocedure has been simulated, overhead is reduced because the abstract step is treated like a primitive step. Another source of efficiency is hierarchical procedures. The hierarchy allows instructors to create relatively small and modular procedures, and the run-time overhead of creating small and modular procedures is small.

4.10 Related Work

4.10.1 Natural Language Versus Direct Manipulation

When using Diligent, the instructor demonstrates a procedure by directly manipulating the environment. However, a natural language (e.g. English) could have been used to specify the procedure's steps. Humans find natural languages flexible and easy to use. Unfortunately, computers have difficulty understanding natural languages. One problem is ambiguity. For example, what does “it” or “the button” mean? Another problem is indirection. Instead of simply performing an action, a human needs to provide an abstract description. A system that receives demonstrations with a similar content as Diligent's, but in English, is Instructo-Soar [HL95].

Although direct manipulation avoids many of the problems of ambiguity inherent in natural languages, a problem when using direct manipulation is handling abstraction. Input is very concrete because it deals with individual objects. This raises the issue of how to specify quantification, negation and sets [Coh92]. Because input is directly entered in the current state, it is also difficult to specify hypothetical situations. These are areas where natural language could complement direct manipulation.

4.10.2 Programming By Demonstration

This section will discuss related work in basic techniques for *Programming By Demonstration* (PBD) [C+93]. A PBD system learns how to perform some task by observing a user perform it. The difference between PBD and learning a macro is that PBD involves a generalization of the task instead of a rote repetition of the user's actions. Diligent can be classified as a PBD system because it observes demonstrations and uses them to create plans and operators.

Many PBD systems learn how to perform procedures. These systems typically utilize a helpful user in order to learn how to perform simple procedures after only a few demonstrations. Diligent differs from a typical PBD system because it has the ability to experiment[18] and the ability to learn the relationships between steps (i.e. step relationships). Additionally, few PBD systems can learn hierarchical procedures.

4.10.2.1 Procedure Representation

An important aspect of this chapter is that it provides algorithms that transform demonstrations into hierarchical partially ordered plans. This plan representation has fine-grained ordering constraints and causal links that have been shown to useful for providing good explanations to humans [ME89].

Some robotic PBD work [FMD+96] has also produced hierarchical partially ordered plans, but the robotic work learns a very different representation. Each step in a procedure has a set of disjunctive preconditions that indicate when the step can be performed if zero to all of the procedure's later steps are not performed. If a procedure is long, then these preconditions could become very complicated. Thus, it appears that human users could have difficulty understanding or verifying the preconditions of steps.

[18]Diligent experiments by replaying a procedure, skipping a step and observing the result.

4.10.2.2 Basic Techniques

The PBD literature provides a number of useful basic techniques. Other than some preliminary work on sensing actions, Diligent used basic PBD techniques instead of creating new ones.

One basic technique is focusing the agent's attention. This can be done by pointing to objects [Mau94] and identifying important objects by performing extraneous actions on them [Lew92]. Using extraneous actions probably has limited value for Diligent because apparently extraneous actions are likely to indicate either that one of the actions is a sensing action, that Diligent cannot see a relevant attribute, or that Diligent is missing knowledge of step relationships. Other work on focusing has looked spatial distance and how quickly actions are performed [Hei89]. However, using spatial distance may have little value on a device with buttons and switches. Using speed of instruction as focus may also be inappropriate because hurrying an instructor may negatively impact the quality of a demonstration.

Besides focusing, another basic PBD technique is asking the user to provide clarification. The user is asked to select between a set of hypotheses in the PRODEGE+ graphics editor [BS93]. In contrast, Metamouse asks the user to toggle on "thumbtacks" which indicate potentially important features [MW93]. Diligent uses this technique when it presents an instructor with hypothesized preconditions, goal conditions and step relationships.

Another technique is providing a graphical history or storyboard [KF93]. A *graphical history* shows in a sequence of small windows how the window used for instruction varied throughout a demonstration. One problem with graphical histories is that support for graphical histories might need to be explicitly designed into a graphical interface. Diligent could not use graphical histories because it did not have enough control over the environment's graphical interface.

One basic technique is learning hierarchical procedures [KM93]. This promotes reuse because existing procedures can be used as components of larger procedures. This improves scalability because it takes less work by a user to enter a large procedure. Diligent's hierarchical procedures are unusual because of the causal links in subprocedures. Causal links provide a great deal of flexibility because they indicate which steps are necessary when starting from a variety of initial states.

Still another basic technique is adding textual annotations (e.g. an object's name) to a graphical representation [Lie94]. A problem with the annotation approach is that Diligent does not have the ability to insert text into the environment's graphical interface. If

Diligent had this ability, it might be able to more clearly communicate with the instructor, but it is unclear how much effect annotations would have.

Another technique is creating graphical rewrite rules. A *graphical rewrite rule* transforms a graphical pattern into another pattern. This technique works best when the user can create the new pattern by making fine-grain changes to a graphical environment. A system that uses this approach is KidSim [SCS94, CS95], which allows young children to create simulations. Diligent does not use this technique because it is designed to be used in environments where it has limited control of the environment.

Work by Bimbo and Viario has addressed an issue that Diligent does not consider, which is training multiple agents in a virtual environment [BV96]. They do this by having all but one agent replay a fixed sequence of actions. The system learns how to react to situations based on spatial and temporal constraints. However, the system does not learn the knowledge necessary for teaching. An issue with this approach is synchronizing the actions of all agents. Synchronization is a problem because the agent and the instructor may be engaged in dialog that conflicts with the time line. Synchronization is also an issue because an agent's actions could cause another agent to deviate from the fixed sequence of actions being replayed.

While the other basic PBD techniques used by Diligent have been discussed in the PBD literature, no other system appears to incorporate actions that actively gather information (i.e. sensing actions) and then use this information to influence a procedure's flow of control. Since Diligent learns procedures for the types of domains where test results are gathered, sensing actions are important.

Most PBD systems merely accept the data provided by the user, but some systems actively identify data that can be used to refine its knowledge. A system is said to engage in *active* learning when it identifies data that can help refine its knowledge. One such system is Disciple [TK98], which finds an example and asks the user whether it belongs to a given class. However, other than Diligent, there appears to be no system that uses direct manipulation and then uses the environment to perform experiments that will reduce the need for the user to answer questions.

4.11 Summary

The main importance of this chapter is that it provides algorithms that transform demonstrations into hierarchical partially ordered plans. While many of the algorithms are original, they tend to be fairly simple or derived from standard planning techniques. This chapter is also important because its algorithms create the basic structure used to learn operators (Chapter 5) or to perform experiments (Chapter 6).

We will now briefly review what this chapter covered.

This chapter discussed how Diligent transforms demonstrations into procedures. To process a demonstration, Diligent can combine multiple demonstrations into a path. Because the path contains all the procedure's steps, Diligent uses the path to derive the procedure's plan. By default, a procedure's goal conditions contain the final values of attribute's whose values changed during the procedure. Once the goals are known, step relationships can be derived using the path's sequence of steps and the preconditions and state changes of each step.

To promote scalability, modularity and ease of authoring, procedures can be hierarchical. Subprocedures can be specified by inserting existing procedures into a demonstration or by creating a new subprocedure inside a demonstration of the parent procedure. However, when reusing an existing procedure as a subprocedure, Diligent needs to internally simulate the subprocedure because the subprocedure's initial state may require skipping some of the subprocedure's steps.

Another issue is how to incorporate sensing actions into a procedure. Because sensing actions do not change the environment, Diligent needs to ensure that they are not skipped. Diligent does this by creating a mental attribute that doesn't exist in the environment and then using the mental attribute in a goal condition. Diligent also ensures that a sensing action is performed in the proper state by adding preconditions that control when it is performed.

Chapter 5

Learning Operators

The previous chapter discussed constructing procedures from demonstrations. However, demonstrations, by themselves, are not useful because they do not explicitly indicate the dependencies between steps (i.e. step relationships). Without knowledge of dependencies, an automated tutor could perform the procedure by rote, but could not answer questions about which steps to perform or how steps depend on each other.

Diligent corrects for this problem by learning operators. An *operator* models actions performed in the environment by indicating which preconditions will cause an action to produce given state changes. Diligent associates the operators that it learns with the steps of procedures. This allows Diligent to use operator preconditions when calculating the dependencies between steps.

One of Diligent's contributions is how it balances the techniques used to learn operators with how it performs experiments. Experiments, which will be discussed in the next chapter, can more easily remove unnecessary preconditions than identify missing ones. In contrast, the techniques that Diligent uses to learn operators have a bias favoring likely but potentially unnecessary preconditions. This bias is important because little data may be available for learning. Part of this bias is Diligent's novel focus on the heuristic that the state changes of earlier steps in a demonstration are likely to establish preconditions for later steps. This heuristic is used when creating new operators.

This chapter discusses how Diligent learns operators. First, we will present requirements for the learning problem. We will then discuss heuristics and data structures. Afterwards, we will discuss how to create new operators and refine existing operators. The chapter will finish with a discussion of run-time complexity and related work.

5.1 Additional Requirements

Earlier in section 3.1, we described the authoring problem in terms of requirements, constraints and the interface to the environment. Since then, the discussion of how demonstrations are processed has made the problem more constrained and concrete. Factors that have constrained the problem include the procedural representation (i.e. plans), how operators are used generate plans, and the number and types of demonstrations provided by the instructor. These additional constraints allow us to define additional requirements that focus on the problem of learning operators.

Most of these additional requirements arise from the general requirements to make the instructor's job easier and to maximize the utility of a procedure's few demonstrations.

The new requirements are as follows.

Requires very little domain knowledge. Diligent may start with no domain knowledge. This means that the learning algorithm cannot rely on detailed domain knowledge.

Quick competence because few action-examples. Diligent needs to find reasonable preconditions quickly because it may have seen only a few demonstrations. If Diligent can find reasonable preconditions, then the instructor's job should be easier.

Incremental or appear incremental. The learning algorithm needs to appear incremental for a number of reasons. First, the data arrives incrementally. Second, instructors would be confused if preconditions looked very different each time an operator was updated. Third, because Diligent is interactive, the algorithm cannot perform slow batch processing.

Support error recovery. Because there needs to be quick competence and because learning is incremental, early preconditions may be incorrect. Thus, Diligent needs to be able to recover from errors that could include both missing and unnecessary preconditions.

Humans can understand the precondition representation. An instructor needs to understand and verify preconditions. Unless the preconditions are concise and explicit, he will not be able to do so. An instructor must also be able to determine whether or not a specific condition is a precondition.

One issue is what representations could an instructor understand. This is a difficult question because there are degrees of understandability. There is evidence that humans have difficulties with some types of simple logical statements [New90]. Because preconditions are a type of logical statement, we will give the intuitive argument that simpler representations should be easier to understand. We are also going to argue that, to avoid problems, the representation should be as simple as feasible.

As an example, consider turning on a car's engine by turning the key. The preconditions for this might be that the key is in the ignition, the seat belt is fastened, and the door is closed. Two ways of representing these preconditions are shown in Figure 5.1. The conjunctive representation used in a) would be used by Diligent and anecdotally appears similar to what humans would use. In contrast, humans appear unlikely to use b), which might be learned by CDL [She93].

a) (keyLocation ignition) $\wedge$ (seatBelt fastened) $\wedge$ (door closed)

b) (keyLocation ignition) $\wedge$ ($\neg$(seatBelt open) $\vee$ (door closed))

Figure 5.1: Preconditions for Starting a Car

Important attributes need to be identified The environment may have hundreds, if not thousands, of attributes, and in a given procedure, most attributes will probably not change value and will probably be irrelevant. Therefore, the learning algorithm needs to help distinguish important attributes from unimportant attributes. In contrast, the learning algorithm could also have required generalizing object classes and replacing attribute values by variables.

Bound a precondition's uncertainty. The instructor should receive some indication of the system's certainty about whether a condition is or is not a precondition. By indicating its confidence in a preconditions, Diligent can help focus the instructor's attention on areas of uncertainty.

5.2 Heuristics

The algorithms in this chapter use some of the heuristics from chapter 3: 1) focus on attributes that change value; 2) the state changes produced by earlier steps are likely to be preconditions of later steps; and 3) favor existing knowledge and hypotheses.

This chapter also uses a new heuristic.

Prefer extra preconditions over missing ones. In the algorithms that will be used, it is easier to remove an invalid precondition than to identify a missing precondition. It should also easier for humans to spot a mistake among a few proposed preconditions than from a large set of unused conditions.

5.3 About this Chapter's Examples

Like the other chapters, this chapter's examples are taken from the HPAC domain. The domain has been simplified in order to illustrate the algorithms. Despite the similarity, the examples in this chapter do not correspond to the extended example of Chapter 4.

5.4 Data Structures

The relevant data structures are the learning algorithm's input and output. The inputs are action-examples and demonstrations, and the outputs are operators.

The action-examples used for learning operators were defined in Section 3.2.1.1. An action-example records the state of the environment before and after an action is performed. The state before the action is called the *pre-state*, and the state after is called the *post-state*. The part of the post-state that changes is called the *delta-state*. States are composed of conjunctive sets of conditions. A condition contains an attribute and its value. For example, the condition (valve1 open) means that attribute valve1 has the value open.

The current demonstration is also used when creating new operators. Demonstrations were defined in Section 4.3. Demonstrations contain a sequence of steps, each of which is associated with an action-example.

The representation of operators was defined in Section 3.2.2.2, but because this chapter focuses on learning operators, we will spend some time discussing and motivating the representation.

Operators model how actions performed by the instructor in the environment affect the state of the environment. Operators identify the preconditions necessary for an action to produce a given set of state changes. Because an action may produce different state changes in different states, an operator's preconditions and state changes are described by one or more conditional effects. Each *conditional effect* (or *effect*) has its own set of preconditions and state changes. Preconditions and state changes are described by conjunctive sets of conditions. When the preconditions are all satisfied in an action-example's pre-state, the associated state changes should be observed in the post-state.

Let c be a condition

c $\in$ g-rep $\Rightarrow$ c $\in$ h-rep $\wedge$ c $\in$ s-rep
c $\in$ h-rep $\Rightarrow$ c $\in$ s-rep

Let S_G, S_H and S_S be the set of environment states
that satisfy the g-rep, s-rep and s-rep, respectively.

$S_S \subseteq S_H \subseteq S_G$

Figure 5.2: Relationship between the Precondition Concepts

Effects have three sets of preconditions (or precondition concepts). In keeping with the terminology used by Wang [Wan96c], the precondition sets are called the s-rep, h-rep and g-rep. However, Wang only used a s-rep and g-rep. The relationship between precondition sets is shown in Figure 5.2. The most specific precondition, *s-rep*, is a superset of the other preconditions. Because the s-rep contains the most conditions, it matches fewer environment states. The heuristic, best guess precondition (*h-rep*) is a subset of the s-rep and matches at least as many environment states as the s-rep. The most general precondition, *g-rep*, is a subset of the other sets and matches at least as many states as the other sets. Although effects have three sets of preconditions, Diligent only uses the h-rep when deriving a plan's step relationships.

Figure 5.3 shows an operator. The operator's action-id indicates that the operator models turning handle handle1. The operator only has one effect, which means that only one set of state changes has been seen. In this case, turning the handle shuts valve1. The h-rep and s-rep contain the g-rep's only condition, (valve1 open), while the s-rep contains a condition, (HandleOn valve1), that is absent from the h-rep and g-rep.

5.4.1 Preconditions as a Version Space

Before proceeding, we will discuss the representation of preconditions as three conjunctive concepts.

An obvious question is whether conjunctive concepts can adequately represent preconditions. An examination of more than 30 domains implemented in PRODIGY showed

Action-id: turn handle1

Effect:

Preconditions:

s-rep: (most specific concept)

(valve1 open)(valve2 open)(HandleOn valve1)

h-rep: (intermediate, heuristic concept)

(valve1 open)(valve2 open)

g-rep: (most general concept)

(valve1 open)

State changes:
(valve1 shut)

Figure 5.3: An Operator

that more than 90% of the operators had only conjunctive preconditions. In the remaining 10%, operators with disjunctive preconditions could be split into multiple operators that have conjunctive preconditions ([Wan96c], page 12). Work on PRODIGY [VCP+95] has tended to focus on using general purpose operators for planning, while Diligent focuses on learning a few specific procedures and does not generalize operators across multiple objects of the same class. Therefore, Diligent is less likely than the work on PRODIGY to need disjunctive preconditions.

The idea of having three concepts for each precondition (i.e. s-rep, h-rep and g-rep) is based on Mitchell's version spaces [Mit78]. In a version space, there is a most general concept, G, and a most specific concept, S. G and S correspond to Diligent's g-rep and s-rep, respectively. G and S are used to classify whether an example belongs to a category. In our case, the "category" is an effect's state changes. Examples rejected by S do not belong to the category, and examples accepted by both S and G belong to the category. Ideally, training with action-examples should cause S and G to converge to a single concept.

Unfortunately, version space algorithms have had run-time complexity problems. Mitchell's Candidate Elimination algorithm [Mit78, Mit82] learns conjunctive conditions where G and S may each contain multiple sets of hypothesized conditions. Unfortunately, Haussler [Hau88] shows that S and G can have an exponential size in relation to the number of training examples. The complexity problems can be partially overcome by using Focusing algorithms [BSP85, YPL77], which learn conjunctive tree structured concepts.[1] Focusing allows S to be represented as a single conjunctive concept, but G may still contain many candidate concepts. Haussler [Hau88] shows that G is still exponential. An exponential size G can be avoided by using the INBF algorithm [SR90], which is a Focusing algorithm that represents G as a single concept because G is conservatively specialized. A key idea of INBF, which Diligent uses, is delaying use of training examples until they can be used and discarded. More recently, Hirsh, Mishra and Pitt [HMP97] have identified efficient version space algorithms for more general classes of concepts; they avoid complexity problems by not explicitly storing S and G. Instead, they determine whether classifying an example as an instance of the concept is consistent with the training examples. However, the lack of an explicit G and S prevents the representation from identifying specific attribute values to use as preconditions. Unfortunately, this violates one of our requirements.

Given that there are many version space algorithms, we will select one for comparison. We will look at OBSERVER's algorithms [Wan95, Wan96a, Wan96c] because OBSERVER, like Diligent, learns conjunctive operator preconditions. OBSERVER's algorithm is similar to INBF. However, instead of learning INBF's tree structured concepts, OBSERVER generalizes its precondition concepts by unifying training examples with operators. This unification results in variables being introduced into the operator.

Unlike OBSERVER, Diligent does not introduce variables into operators through unification. OBSERVER's unification algorithm requires explicit relations between objects and their attributes, but Diligent's unstructured environment does not contain these relations. Additionally, Diligent and OBSERVER have different learning tasks: OBSERVER learns general operators for planning, while Diligent learns a few specified procedures in domains where many objects in a given class (e.g. buttons) may have idiosyncratic behavior. For example, one button may turn on the power, while another starts the motor.

Still, Diligent could have provisionally generalized operators to act on objects of the same class. This generalization could then have been withdrawn if an object was shown to have idiosyncratic behavior. However, in domains that Diligent has used, too many

[1]In a *tree structured* concept, concepts lower in the tree are specializations of concepts higher in the tree. For example, birch and elm are specializations of tree and plant.

objects (e.g. buttons and switches) have idiosyncratic behavior for generalization to be an important capability.

Another issue is the convergence of the s-rep and g-rep to a single concept, especially when there are limited numbers of training examples. What if s-rep and g-rep don't converge? Which one should be used as the precondition? The g-rep is likely to be too general, while the s-rep is likely to be too specific. Choosing between s-rep and g-rep is especially problematic immediately after the version space is created; the s-rep and g-rep are useless because the s-rep matches only one state and the g-rep matches any state. This issue is complicated by the fact that Diligent is unlikely to get enough examples for the s-rep and g-rep to converge.

To avoid problems with version space convergence, Diligent creates plans using the h-rep, which is an heuristic, best guess precondition. The h-rep, which is not present in OBSERVER, is more specific than the g-rep and more general than the s-rep. Thus any state that satisfies the s-rep also satisfies the h-rep, and any state that satisfies the h-rep also satisfies the g-rep.

The h-rep serves a number of purposes. The h-rep provides a usable precondition when there isn't enough data to make the s-rep and g-rep usable. The h-rep also provides a working hypothesis to actively investigate. The idea of a working hypothesis is apparent when you view the three precondition concepts as representing sufficient (s-rep), likely (h-rep) and necessary (g-rep) preconditions.

Even though Diligent uses the h-rep, the s-rep and g-rep are still valuable. As will be shown, the s-rep and g-rep can be used to detect problems with the learning algorithm. The s-rep and g-rep are also used to add missing conditions to the h-rep.

5.5 Creating a New Operator

Figure 5.4 shows the algorithm for creating a new operator. The current demonstration and the action-example of the new operator's action are used to create the operator's first effect. On line 2, g-rep is set to the empty set, and on line 3, s-rep is set to the action-example's pre-state. Thus, g-rep is satisfied by any state, and s-rep is only satisfied by the pre-state. At this point, the g-rep and s-rep are not very useful, but they do bound uncertainty in the preconditions. On line 4, the initial h-rep is set to the pre-state values of attributes that have changed value during the demonstration.[2] Line 5 gathers the pre-state

[2]The algorithm for **Compute-Changes-in-Demo** is in section 4.7.4 on page 4.7.4.

```
procedure Create-New-Operator

Given:    demo: A demonstration.
          ex: An action-example.
Learn:    op: A new operator.

1. Create operator op with effect eff
2. g-rep(eff) ← ∅
3. s-rep(eff) ← pre-state(ex)
4. h-rep(eff) ← Compute-Changes-in-Demo with demo and pre-state(ex).
        (This identifies attributes that have already changed value in the
        demonstration.)
5. h-rep-cand ← conditions in pre-state(ex) that have the same attributes as
        conditions in delta-state(ex).
        (Each condition $c_1$ such that $c_1 \in$ pre-state(ex) and there exists a
        condition $c_2 \in$ delta-state(ex) where attribute($c_1$) = attribute($c_2$).)

6. h-rep(eff) ← h-rep(eff) ∪ h-rep-cand
7. state-changes(eff) ← delta-state(ex)
```

Figure 5.4: Algorithm for Creating New Operator

conditions of attributes whose value changed in the action-example, and line 6 adds these conditions to the h-rep. Because the h-rep reflects changes during the demonstration, the h-rep is a better initial precondition than either the g-rep or the s-rep. Finally, on line 7, the effect's state changes are set to the action-example's delta-state, which contains the post-state values of attributes whose values were changed by the action.

This approach has some similarity with the method used by Instructo-Soar [HL95] to induce conditions under which an action should be performed. Instructo-Soar looks at two groups of conditions: the first group contains the attributes whose values were changed by the action, and the second group contains relations between the objects being acted upon and the objects associated with the procedure's goal conditions.

In contrast, the preconditions of Diligent's operators attempt to model the environment in a way that is independent of a given step or procedure. That is why Diligent doesn't need the procedure's goals when learning preconditions and that is why Diligent looks at the state changes of the demonstration's earlier steps, which are likely to be preconditions of later steps.

Demonstration:
 demo1, with the following state change earlier in the demonstration
  (HandleOn valve1)

Action-example:
 Action-id:
  turn handle1
 Pre-state:
  (valve1 open)(valve2 open)(valve3 open)(HandleOn valve1)
 Delta-state:
  (valve1 shut)

Figure 5.5: Input for Creating New Operator

Figure 5.5 shows the input for creating a new operator, and Figure 5.6 shows the resulting operator. In Figure 5.5, the only state change from earlier steps in the demonstration is that the handle was moved to valve1 ((HandleOn valve1)); this condition is added to the new operator's h-rep by line 4 of Figure 5.4. Additionally, the only attribute in the delta-state (valve1) has its pre-state condition (valve1 open) added to the h-rep by line 5 of Figure 5.4.

Operator: turn-handle

Action-id:
 turn handle1

Effect 1:
 Preconditions:
  g-rep:
   $\emptyset$
  h-rep:
   (valve1 open)(HandleOn valve1)
  s-rep:
   (valve1 open)(valve2 open)(valve3 open)(HandleOn valve1)
 State changes:
  (valve1 shut)

Figure 5.6: A New Operator

5.6 Positive and Negative Examples

Desired state change:
(valve1 open)

Positive example:

Pre-state:	**Post-state:**	**Delta-state:**
(valve1 closed)	*(valve1 open)*	*(valve1 open)*

Negative example:

Pre-state:	**Post-state:**	**Delta-state:**
(valve1 closed)	(valve1 closed)	$\emptyset$

Indeterminate:

Pre-state:	**Post-state:**	**Delta-state:**
(valve1 open)	*(valve1 open)*	$\emptyset$

Figure 5.7: Some Positive and Negative Examples

Because Diligent may receive little input, it needs to learn quickly. One way of learning faster is to learn from both success and failure. Success means that an action produces the desired result, and failure means that an action doesn't produce the desired result. Diligent learns from success and failure by comparing an operator's effects to action-examples.

To use an action-example, each action-example's action-id is matched with the operator that models that action. The action-example is only used to refine that one operator.

To refine one of the operator's effects with the action-example, Diligent uses the common machine learning technique of classifying each action-example as either a positive or negative training example. A *positive example* contains the effect's state changes in its delta-state, and a *negative example* does not contain the effect's state changes in it post-state. It is *indeterminate* whether an action-example should be classified as either positive or negative if the action-example contains the effect's state changes in both its pre-state and post-state. It is indeterminate because it is unknown whether the action did not change the attributes in the effect's state changes or whether the action did change the values but to their pre-state values. Figure 5.7 illustrates how to classify examples for an effect that opens valve1. In negative example, the attribute in the effect's state change (valve1) doesn't have the desired value in the action-example's post-state. In **indeterminate example**, the attribute has the desired value in both the pre-state and post-state, and Diligent only looks at attributes that clearly changed value (i.e. are in the delta-state).

5.7 Refining Preconditions

Once action-examples have been classified, Diligent uses the techniques of Incremental Non-Backtracking Focusing (INBF) [SR90] to generalize precondition concepts with positive examples and specialize precondition concepts with negative examples. The most specific concept (s-rep) is generalized if it incorrectly classifies a positive example. The s-rep is generalized by removing attributes whose pre-state values don't match the values in the s-rep. The most general concept (g-rep) can be specialized if the g-rep incorrectly classifies a negative example. The g-rep is specialized by adding a condition from the s-rep whose attribute has a different value in the s-rep than in the negative example's pre-state. Because the g-rep now contains an additional condition, it can correctly classify the example as negative. Because of the difficulty identifying which condition to add, the g-rep is only updated if there is a near-miss between the s-rep and the negative example. There is a *near-miss* when only one s-rep condition does not match the negative example's pre-state. Requiring a near-miss is a conservative approach that only adds conditions to the g-rep when they have been shown to be necessary. Because a negative example may not be a near-miss, a negative example is kept until it achieves a near-miss or the g-rep correctly classifies it as negative. In a similar manner, the h-rep can be generalized like the s-rep or specialized like the g-rep.

Because the g-rep and s-rep provide an upper and lower bound for the h-rep, the h-rep doesn't have to be updated as conservatively as the g-rep and s-rep. The g-rep and s-rep are conservatively updated because they represent the most general and most specific candidate preconditions. Because the g-rep is only specialized and the s-rep is only generalized, changes to the g-rep and s-rep cannot be undone. In contrast, the h-rep has a capacity for error recovery since it can be both specialized and generalized. Error recovery may be necessary for the h-rep because it only represents a "best" working hypothesis.

In the following sections, we will look at refining preconditions with positive and negative examples.

```
procedure Refine-Positive-Example

Given: op: An operator.
       eff: An effect of op.
       ex: An action-example that is a positive example of eff.
Learn: Refined preconditions for eff.

1. collapse-list ← $\emptyset$
2. diff ← s-rep conditions whose attributes
       have different values than in the action-example's pre-state.
       (The conditions in diff appear unnecessary.)
3. For each condition cond $\in$ diff,
       a) If cond is in the effect's g-rep, then add cond to collapse-list.
4. If collapse-list $\neq \emptyset$ then
       a) The version space has collapsed, and the elements of collapse-list
          appear to be incorrect. Ask the instructor to update
          the preconditions of eff using collapse-list.
       b) Return
5. Remove from s-rep any conditions contained in diff.
6. Remove from h-rep any conditions contained in diff.
7. For all unused negative examples neg-ex of eff,
       a) Use Refine-Negative-Example on op and eff with neg-ex.
          (Because conditions have been removed from s-rep, there
          are fewer conditions that could distinguish positive and
          negative examples.)
```

Figure 5.8: Refining Preconditions with a Positive Example

5.7.1 Refining Preconditions with Positive Examples

Positive action-examples are used to remove unnecessary conditions from an effect's precondition concepts. The algorithm for refining an effect's preconditions with a positive action-example is shown in Figure 5.8.[3] To process an example, we need to to identify unnecessary preconditions. This is done on line 2, which identifies conditions in the most specific precondition concept (s-rep) that do not match the pre-state of the action-example.[4] The unnecessary conditions from line 2 are removed from the s-rep and h-rep on lines 5 and 6.

[3]For clarity, some minor efficiency improvements have been removed.

[4]We assume that no attributes were added to or removed from the state.

Besides removing unnecessary conditions, we need to check that the preconditions are consistent with the training data; this is done in lines 3 and 4. A key idea is that the g-rep's conditions have already been shown to be necessary. Line 3 identifies necessary conditions that now appear unnecessary, and line 4 indicates an interaction with the instructor to correct the problem.

When a condition is shown to be both necessary and unnecessary, the version space is said to *collapse*. There are several reasons for a version space to collapse: 1) the instructor introduced errors when editing preconditions; 2) Diligent cannot see a necessary environment attribute; or 3) the precondition needs to be represented as a disjunction of conjunctive conditions. All three of these cases need further interaction with the instructor and are beyond the scope of our present discussion.

After unnecessary preconditions have been removed by lines 5 and 6, the differences between the preconditions and negative examples might be smaller. For this reason, the effect is checked against negative examples that previously produced far-misses (line 7). A *far-miss* indicates that two or more attributes in the s-rep have different values than in the action-example's pre-state.

Positive example pre-state:
(valve1 open)
(valve2 open)
(valve3 shut)
(HandleOn valve1)
(AlarmLight1 off)

Preconditions before:	**Preconditions after**:
g-rep:	**g-rep**:
(valve1 open)	(valve1 open)
h-rep:	**h-rep**:
(valve1 open)	(valve1 open)
(valve3 open)	(HandleOn valve1)
(HandleOn valve1)	
s-rep:	**s-rep**:
(valve1 open)	(valve1 open)
(valve2 open)	(valve2 open)
(valve3 open)	(HandleOn valve1)
(HandleOn valve1)	

Figure 5.9: Using a Positive Example

The algorithm for processing positive examples (Figure 5.8) is illustrated by Figure 5.9. Line 2 computes the set of differences between the s-rep and the action-example (*diff*). In this case, the set contains (valve3 open) but not (AlarmLight1 off). The condition (AlarmLight1 off) is ignored because the s-rep doesn't contain a condition involving the attribute AlarmLight1. The version space would collapse (lines 3 and 4) only if (valve1 open) was not in the action-example's pre-state. On lines 5 and 6, the condition (valve3 open) is removed from the s-rep and h-rep.

5.7.2 Refining Preconditions with Negative Examples

Before proceeding, we will discuss potentially needed conditions, which are derived from INBF [SR90]. Potentially needed conditions are defined in Figure 5.10.[5] At least one of the potentially needed condition must distinguish a given negative example from positive examples. In the HPAC domain, there are several dozen conditions in an action-example's pre-state, but usually only a few potentially needed conditions. There are so few conditions because Diligent focuses on learning the procedures specified by the instructor rather than exploring the environment, and this creates a tendency for positive and negative examples to have similar pre-states.

Potentially-Needed-Conditions $\equiv$ s-rep conditions whose attributes have different values in an action-example's pre-state.

$$\equiv \{\ c_1 \mid c_1 \in \text{s-rep} \wedge c_2 \in \text{pre-state} \wedge \text{attribute}(c_1) = \text{attribute}(c_2) \wedge \text{value}(c_1) \neq \text{value}(c_2)\ \}$$

Figure 5.10: Potentially Needed Conditions

Negative examples are used to add conditions to an effect's preconditions. This is done by looking for a one condition or near-miss mismatch between the s-rep and an action-example's pre-state. The algorithm is shown in Figure 5.11[6] and will be illustrated by the action-examples in Figure 5.12. Note that action-examples are added to a set of unused negative examples (line 2) and then removed when nothing more can learned from them (lines 4a and 6b).[7]

[5]Instead of potentially needed conditions, INBF used potentially guilty conditions, which contain conditions from the example's pre-state rather than the effect's s-rep.

[6]In Diligent, incrementally storing and updating potentially needed conditions greatly reduced the number of conditions checked. However, for clarity, these simple changes to the algorithms are not shown.

[7]Because the s-rep is used to identify potentially needed conditions, both the s-rep and g-rep are necessary for identifying missing preconditions.

```
procedure **Refine-Negative-Example**

Given: op: An operator.
       eff: An effect of op.
       ex: An action-example of eff.
Learn: Refined preconditions for eff.

1.
1. If state-changes(eff) $\subseteq$ post-state(ex) then
   (This is true when state-changes(eff) $\subseteq$ pre-state(ex).)

      a) return (The example should be classified as indeterminate rather
             than negative.)

2. Add ex to the set of unused negative examples of eff.
   (Keep ex until it is rejected by the g-rep(eff).)
3. needed-cond $\leftarrow$ Potentially-Needed-Conditions of ex for eff
   (These conditions distinguish ex from positive examples.)
4. If needed-cond $\cap$ g-rep(eff) $\neq \emptyset$ then
      a) Nothing can be learned from ex because g-rep(eff) classifies
         it as negative. Remove ex from the set of unused negative examples.
      b) return
5. If needed-cond $= \emptyset$ then
      a) collapse-list $\leftarrow$ conditions in eff's original s-rep that are not in the
         current s-rep.
      b) Some of the conditions in collapse-list are required, ask
         the instructor to update the preconditions of eff using collapse-list.
      c) return
6. If needed-cond has only one condition then
      a) Add the condition to eff's g-rep and h-rep.
      b) Nothing more can be learned from ex. Remove it from the set of
         unused negative examples.
      c) return
7. If needed-cond $\cap$ h-rep(eff) $\neq \emptyset$ then
      a) return
         (h-rep(eff) classifies the ex as negative, but we are uncertain which
         conditions distinguish ex from positive examples.)
8. h-rep(eff) classifies the ex as a positive example. Attempt to refine h-rep(eff)
   with ex by invoking Discriminate-With-Other-Effects.
```

Figure 5.11: Refining Preconditions with Negative Example

Action-example 1:
- **Pre-State:**
  - (valve1 shut)
  - (valve2 open)
  - (valve3 open)
  - (HandleOn valve1)
- **Delta-State:**
  - (valve2 shut)

Action-example 2:
- **Pre-State:**
  - (valve1 open)
  - (valve2 open)
  - (valve3 open)
  - (HandleOn valve2)
- **Delta-State:**
  - (valve2 shut)

Action-example 3:
- **Pre-State:**
  - (valve1 open)
  - (valve2 open)
  - (valve3 shut)
  - (HandleOn valve2)
- **Delta-State:**
  - (valve2 shut)

Action-example 4:
- **Pre-State:**
  - (valve1 shut)
  - (valve2 shut)
  - (valve3 open)
  - (HandleOn valve1)
- **Delta-State:**
  - (valve1 open)

Effect:
- **State changes:**
  - (valve1 shut)

- **Preconditions before:**
  - **g-rep:**
    - $\emptyset$
  - **h-rep:**
    - (valve1 open)
  - **s-rep:**
    - (valve1 open)
    - (valve2 open)
    - (valve3 open)
    - (HandleOn valve1)

- **Preconditions after:**
  - **g-rep:**
    - *(HandleOn valve1)*
  - **h-rep:**
    - (valve1 open)
    - *(HandleOn valve1)*
  - **s-rep:**
    - (valve1 open)
    - (valve2 open)
    - (valve3 open)
    - (HandleOn valve1)

Figure 5.12: Using Negative Examples

Action-example 1 is rejected by line 1 of the algorithm because Diligent cannot determine whether it is a negative or positive example. The effect's state change, (valve1 shut), is satisfied in the action-example's pre-state and post-state. Diligent cannot determine whether attribute valve1's value was constant or was changed back to the attribute's pre-state value. Because Diligent cannot correctly classify the action-example as either positive or negative, using the action-example could introduce errors into the effect's preconditions.

Action-example 2 adds a condition to the g-rep and h-rep. The preconditions before and after processing the action-example are shown on the bottom of Figure 5.12. On line 3, only one potentially needed condition is found ((HandleOn valve1)). Since the condition is not part of the g-rep, the g-rep misclassifies the condition as positive (line 4). Since there is only one potentially needed condition, line 6 specializes the g-rep and h-rep by adding the condition to them. At this point, the algorithm cannot learn anything more from the action-example, and the action-example is removed from the set of unused negative examples.

Action-example 3 is rejected because the g-rep correctly classifies it as a negative example. Line 3 identifies the potentially needed conditions ({(valve3 open) (HandleOn valve1)}). Line 4 then checks if any of these conditions are in the g-rep. One of the conditions ((HandleOn valve1)) is in the g-rep. At this point, the action-example is rejected because nothing can be learned from it. On line 4a, the action-example is removed from the set of unused negative examples.

Action-example 4 is rejected by the h-rep but not by the g-rep. On line 3, two potentially needed conditions ({(valve1 open)(valve2 open)}) are found. The test on line 4 fails because neither condition is in the g-rep.[8] Because there is more than one potentially needed condition, no condition is added to the g-rep and h-rep (line 6). Finally, on line 7, the action-example is rejected because the h-rep condition (valve1 open) is also one of the potentially needed conditions. However, unlike action-examples 2 and 3, action-example 4 remains in the set of unused negative examples because it can still be used for identifying preconditions as necessary (i.e. in g-rep).

Line 5 deals with the collapse of the version space. The version space collapses when a condition needed for distinguishing between positive and negative examples has been shown to be unnecessary. The reasons for a collapse were discussed in Section 5.7.1.

[8]Since the effect's state change is (valve1 shut), one might expect (valve1 open) to be in the g-rep. However, condition (valve1 open) hasn't been shown to be necessary, and attribute valve1 potentially could have many values.

Line 8 is used when the h-rep misclassifies a negative example as a positive. In this case, the h-rep will be compared to preconditions in the operator's other effects. Although a condition might be added to the h-rep, no condition will be added to the g-rep. This processing will be discussed in the next section.

5.7.2.1 Discriminating Between Effects

There is an additional opportunity to learn when an effect's h-rep misclassifies a negative example as positive. In this situation, the action-example always has at least two potentially needed conditions, but none of them are in the h-rep. At least one of these potentially needed conditions should be in the h-rep.

Fortunately, the operator's other effects are likely to have similar preconditions. That is because the preconditions need to differentiate between situations where different state changes are observed, especially when two effects cause the same attribute to have different values. For example, consider a button that toggles whether the power is on or off. The preconditions for turning the power on need to reject every pre-state where pressing the button will turn the power off.

This means that we might identify a precondition by examining the preconditions of other effects. In particular, we are interested incompatible effects. Two effects are *incompatible* if they have a state change for the same attribute but with different values. Comparing incompatible effects is a reasonable approach because their preconditions must differentiate their state changes.

When comparing incompatible effects, Diligent requires the action-example to be positive for one incompatible effect and negative for the other. We will call the effect with the negative example N and the effect with the positive example P. Diligent adds a condition to effect N's h-rep when there is a near-miss between N's potentially needed conditions and effect P's preconditions. Requiring a near-miss provides more evidence for the condition; without this evidence, an attribute in one effect's preconditions might get unnecessarily added to all the others. When looking for a near-miss, Diligent checks all three of P's precondition concepts (i.e. s-rep, h-rep and g-rep).

The algorithm in Figure 5.13 will be illustrated with the action-example in Figure 5.14.

Recall from the previous section that procedure **Refine-Negative-Example** invokes procedure **Discriminate-With-Other-Effects** when the h-rep misclassifies a negative example as positive. Unless the instructor had edited the preconditions, this can only happen if an attribute in the effect's state changes can take three or more values because, by default, the pre-state values of attributes in the state change are in the h-rep. The

```
procedure **Discriminate-With-Other-Effects**

Given:  *op*: An operator.
        *eff*: An effect of *op*.
        *ex*: An action-example that is a negative example of *eff*.
Result: Refine effect *eff*'s h-rep

1. For each incompatible effect (*incomp-eff*) of effect *eff* for operator *op*,

   Effect *incomp-eff* is incompatible when there exists conditions $c_1$ and
   $c_2$ such that $c_1 \in$ state-change(*eff*) $\wedge$ $c_2 \in$ state-change(*incomp-eff*) $\wedge$
   attribute($c_1$) = attribute($c_2$) $\wedge$ value($c_1$) $\neq$ value($c_2$).

        a) If *ex* is a positive example of *incomp-eff*, then attempt to refine
           h-rep(*eff*) with **Discriminate-Between-Effects**.


procedure **Discriminate-Between-Effects**

Given:  *eff*: An effect.
        *incomp-eff*: An effect that is incompatible with *eff*.
        *ex*: A negative example of *eff* and a positive example of *incomp-eff*.
        *cands*: Candidate conditions for h-rep(*eff*). These are the potentially
           needed conditions of *ex* for *eff*.
Result: Refine effect *eff*'s h-rep

2. For each of *incomp-eff*'s precondition concepts (*rep*) (i.e. s-rep, h-rep or g-rep)
   do the following

   a) Find all conditions in *cands* that are not in *rep*, but
        have a common attribute with a condition in *rep*. Call this
        set *cands2*.

        *cands2* $\leftarrow \{c_1 \mid c_1 \in$ *cands* $\wedge \exists c_2 \in$ *rep* where
           attribute($c_1$) = attribute($c_2$) $\wedge$ value($c_1$) $\neq$ value($c_2$)$\}$

   b) If *cands2* contains one condition,
           i) Add the condition to *eff*'s h-rep.
           ii) Return
```

Figure 5.13: Discriminating Between Effects

Action-example:
- **Pre-state:**
  - (valve1 open)
  - (valve2 shut)
  - (pressure high)
  - (status test)
- **Delta-state:**
  - (status halted)

Incompatible effect:
- **State changes:**
  - (status halted)
- **Preconditions:**
  - **g-rep:**
    - $\emptyset$
  - **h-rep:**
    - (pressure high)
    - (status test)
  - **s-rep:**
    - (valve1 open)
    - (valve2 shut)
    - (pressure high)
    - (status test)

Effect:
- **State changes:**
  - (status normal)
- **Preconditions before:**
  - **g-rep:**
    - $\emptyset$
  - **h-rep:**
    - (valve1 open)
  - **s-rep:**
    - (valve1 open)
    - *(valve2 open)*
    - *(pressure normal)*
- **Preconditions after:**
  - **g-rep:**
    - $\emptyset$
  - **h-rep:**
    - (valve1 open)
    - *(pressure normal)*
  - **s-rep:**
    - (valve1 open)
    - (valve2 open)
    - (pressure normal)

Figure 5.14: An Example of Discriminating Between Effects

h-rep condition can then be removed if the attribute has a different pre-state value in a positive example.

The procedure **Discriminate-With-Other-Effects** (Figure 5.13) first identifies incompatible effects that merit further processing. Line 1 finds all incompatible effects for which the given action-example is a positive example. For these incompatible effects, procedure **Discriminate-Between-Effects** is invoked. In our example (Figure 5.14), only one appropriate incompatible effect is found.

In procedure **Discriminate-Between-Effects**, the potentially needed conditions (*cands*) of the first effect (*eff*) are compared against the preconditions of the incompatible effect (*incomp-eff*). In the example, the potentially needed conditions are {(valve2 open)(pressure normal)}. When the needed conditions are checked against the s-rep of the incompatible effect (line 2a), both potentially needed conditions match. Because checking the s-rep failed, the h-rep is checked. In this case, the h-rep and the potentially needed conditions have a one condition match. This condition, (pressure normal), is then added to effect *eff*'s h-rep. The updated effect is shown in the lower right portion of Figure 5.14.

Another system that compares the preconditions of different state changes is LIVE [She93, She94], but its algorithm is inappropriate for Diligent. LIVE's learning algorithm, Complementary Discrimination Learning (CDL), corrects for the misclassification of an action-example by adding additional conditions to a potentially complicated set of disjunctive preconditions. Unfortunately, a complicated precondition can be created when a simple one could have expressed the same concept. A problem with CDL is that it creates both disjuncts and negated preconditions. A negated precondition indicates that an attribute cannot have a given value. For example, a normal condition may indicate that valve1 is shut, while a negated condition might indicate that valve1 is *not* open. Because preconditions may be unnecessarily complicated, the preconditions may not be suitable for teaching and may not seem reasonable to a human instructor.[9]

5.8 Putting it all Together

So far we have discussed how to create an operator and its first effect. We have also discussed how to refine an existing effect with positive and negative examples. However, we have not discussed the higher level processing that deals with operators and action-examples.

[9]Figure 5.1 contrasted two preconditions for the same effect. The simple one used Diligent's representation, and the complicated one is typical of what CDL would learn.

The next few sections discuss using an action-example to refine an operator. First, we will cover the high level processing that determines how to treat each effect. Second, we will discuss adding a new effect to an existing operator. Third, we will discuss splitting an effect with multiple state changes into two effects.

5.8.1 Determining How to Process Effects

A comparison between an action-example's delta-state and the state changes of the operator's effects determines the type of processing performed on the action-example. If the state changes match an effect's delta-state, the action-example is positive for that effect; and if the state changes and delta-state don't match, the action-example is negative or indeterminate. However, the match might only be partial. Additionally, some of the delta-state's conditions may not match any effect's state changes. These cases need to be taken into account.

Operators are refined by procedure **Refine-Operator** (Figure 5.15).

For an operator to properly model an action-example, the operator needs to predict all the action-example's delta-state conditions. Diligent does this by matching each delta-state condition with some effect's state changes. Initially, all delta-state conditions are added to the set of unmatched delta-state conditions (the set *delta* on line 1). As each effect is processed, any delta-state conditions that match the effect's state changes are removed from the set of unmatched conditions (line 2b). Finally, if any conditions remain unmatched, a new effect is created that has the unmatched conditions as its state changes (line 3).

To discuss the processing of an effect, we will use the data in Figure 5.16. On line 1 (Figure 5.15), the initially unmatched delta-state conditions are {(valve1 shut)(AlarmLight1 on)(AlarmLight3 on)}. Consider effect 1. All its state changes match the delta-state (line 2a). Thus, the example is positive (line 2c). Consider effect 2. None of its state changes match the delta-state. Thus, the example is negative or indeterminate (line 2d). Consider effect 3. Some of its state changes match ((AlarmLight1 on)), but some do not ((AlarmLight2 on)). Thus, the effect is split into two effects (line 2e). Finally, one delta-state condition ((AlarmLight3 on)) is unmatched by any effect. In this case, Diligent creates a new effect for the unmatched condition (line 3).

```
procedure **Refine-Operator**

Given:   op: An operator.
         ex: An action-example for the operator.
Result:  Refine operator op.

1. delta ← action-example ex's delta-state

2. For each effect eff of operator op,
        a) Identify conditions in eff's state changes that
           match (meff) and do not match (feff) the action-example ex

           meff ← state-changes(eff) $\cap$ delta
           feff ← state-changes(eff) $\setminus$ meff

        b) Remove each condition from delta that matches one of
           the effect's (eff) state changes.

        c) If all state changes match the action-example (feff $= \emptyset$),
              i) Refine effect eff with a positive example ex
                 by invoking Refine-Positive-Example.

        d) Else if no state changes match the action-example (meff $= \emptyset$),
              i) Example ex is either negative or indeterminate.
                 Refine effect eff with example ex
                 by invoking Refine-Negative-Example.

        e) Else the action-example only matches some state changes,
              i) Split the effect eff in two with Split-Effect. Use the
                 action-example ex and the matching (meff) and
                 mismatching (feff) state changes.

3. If some conditions in the action-example's delta-state haven't been matched
   (delta $\neq \emptyset$),
        a) Create a new effect by invoking Create-New-Effect and
           using the action-example ex and the unused delta-state
           conditions delta.
```

Figure 5.15: Refining an Operator with an Example

Action-example:
  Pre-state:
    (valve1 open)(AlarmLight1 off)(AlarmLight2 off)(AlarmLight3 off)
  Delta-state:
    (valve1 shut)(AlarmLight1 on)(AlarmLight3 on)

Effect 1:
  State Changes:
    (valve1 shut)

Effect 2:
  State Changes:
    (valve1 open)

Effect 3:
  State Changes:
    (AlarmLight1 on)(AlarmLight2 on)

Figure 5.16: An Example for Assigning Delta-State Conditions to Effects

5.8.2 Adding a New Effect

In previous sections, we have discussed how to create operators and refine them with action-examples, but we have not discussed adding new effects to existing operators. A new effect is added when no conditions in any existing effect's state changes match some condition in an action-example's delta-state.

When creating a new effect, the assumptions used to create the first effect's preconditions may be inappropriate. For instance, the action-example might occur while Diligent is performing an experiment rather than during a careful constructed demonstration. During a demonstration an instructor is likely to group related steps together so that earlier steps establish preconditions of later steps. In contrast, a new effect might might be seen during an experiment because a precondition of an existing effect was not satisfied. Fortunately, the preconditions of existing effects are good sources of knowledge because they have probably undergone some refinement. Therefore, when an operator already has an effect, Diligent uses the knowledge already contained in the operator rather than the state changes of previous steps.

The algorithm for creating the new effect is shown in Figure 5.17 and will be discussed in the next few paragraphs.

```
procedure Create-New-Effect

Given:  op: An operator, ex: An action-example of that operator, and
        delta: A set of state changes.
Result: Create a new effect for operator op.

1. For operator op, create a new effect new-eff.
2. Set the effect's state changes to delta.
3. Since action-example ex is new-eff's first positive example,
   initialize the version space bounds with ex.

   s-rep(new-eff) ← pre-state(ex) & g-rep(new-eff) ← $\emptyset$

4. Find the operator's earlier action-example (similar-ex) that is most similar to ex.
   Similarity is measured by the fewest differences between action-example pre-states.
5. Find the conditions (h-rep1) in ex's pre-state that are different than
   conditions in similar-ex's pre-state.

   h-rep1 ← { $c_1$ | $c_1 \in$ pre-state(ex) $\wedge$ $c_2 \in$ pre-state(similar-ex) $\wedge$
                attribute($c_1$) = attribute($c_2$) $\wedge$ value($c_1$) $\neq$ value($c_2$) }

6. Select an earlier effect (old-ce) whose h-rep will used to help initialize
   the h-rep for new-eff.
   (Diligent chooses the operator's first effect.)
7. Create a partial h-rep (h-rep2) by making the earlier
   effect's (old-ce) h-rep consistent the action-example's (ex) pre-state.

   h-rep2 ← { $c_1$ | $c_1 \in$ pre-state(ex) $\wedge$ $c_2 \in$ h-rep(old-ce) $\wedge$
                attribute($c_1$) = attribute($c_2$) }

8. Initialize the new effect's best guess precondition concept (h-rep).

   h-rep(new-eff) ← h-rep1 $\cup$ h-rep2

9. For each previous action-example (old-ex) of the operator,
        a) Refine the new effect new-eff by invoking
           Refine-Negative-Example with action-example old-ex.
```

Figure 5.17: Creating a New Effect

The new effect's most general (g-rep) and most specific (s-rep) precondition concepts are initialized with the same method as the operator's first effect. Diligent uses the same method because incorrect conditions cannot be removed from the g-rep and missing conditions cannot be added to the s-rep. Thus, the initial g-rep contains no conditions, and the initial s-rep contains every condition in the action-example's pre-state (line 3).

The initialization of the h-rep exploits knowledge of earlier action-examples and other effects by finding similarities and differences. Although the current action-example is positive for the new effect, all earlier action-examples are negative. Because the h-rep needs to distinguish between positive and negative examples, conditions that distinguish between the current action-example and the closest negative example are likely preconditions (lines 4 and 5).

The initialization of the h-rep also exploits knowledge of other effects by finding similarities between them and the current action-example. Because the preconditions of different effects need to distinguish between various state changes, the attributes used in one effect's h-rep are likely to be useful in the new effect's h-rep (lines 6 and 7). For example, in the HPAC domain, the attribute that indicates which valve a handle is residing on is equally important when opening or shutting the valve.

One problem with using existing preconditions is that they may not be very refined. The lack of refinement can result in missing and unnecessary h-rep conditions. To avoid this problem, the h-rep belonging to the first effect is used because Diligent assumes that the first effect is probably the most refined and accurate (line 6).

Once the new effect has been initialized, Diligent refines the effect with the operator's earlier action-examples. Since the earlier action-examples are all negative or indeterminate, Diligent attempts to add conditions to the new effect's g-rep and h-rep (line 9).

The creation of a new effect is illustrated by Figure 5.18. The "closest earlier action-example" represents *similar-ex* on the algorithm's line 4 (Figure 5.17), and the "first effect" represents *old-ce* on line 6.[10] The differences between the current and previous action-example (*h-rep1* on line 5) are {(HandleOn valve1) (AlarmLight1 on)}. The earlier effect's h-rep and the current action-example's pre-state are compared to produce *h-rep2* on line 7. The set *h-rep2* contains two conditions: one condition, (HandleOn1 valve1), matches the earlier effect's h-rep and one condition does not, (valve1 shut). Finally, the two sets, *h-rep1* and *h-rep2*, are combined on line 8 to form the new effect's h-rep.

[10]Diligent does not care whether *similar-ex* is a positive example of *old-ce*.

Closest earlier example:	**Current example:**
Pre-state:	**Pre-state:**
(valve1 shut)	(valve1 shut)
(valve2 shut)	(valve2 shut)
(HandleOn valve2)	(HandleOn valve1)
(alarm-light1 off)	(alarm-light1 on)
(alarm-light2 off)	(alarm-light2 off)
Delta-state:	**Delta-state:**
(valve2 open)	(valve1 open)
First effect:	**New effect**
State changes:	**State changes:**
(valve1 shut)	(valve1 open)
Preconditions:	**Preconditions:**
g-rep:	**g-rep:**
Don't care	$\emptyset$
h-rep:	**h-rep:**
(valve1 open)	(valve1 shut)
(HandleOn valve1)	(HandleOn valve1)
	(alarm-light1 on)
s-rep:	**s-rep:**
Don't care	(valve1 shut)
	(valve2 shut)
	(HandleOn valve1)
	(alarm-light1 on)
	(alarm-light2 off)

Figure 5.18: An Example of Creating a New Effect

5.8.3 Splitting an Effect in Two

In the previous section, we discussed how to create a new effect from an action-example's delta-state by using conditions that are unmatched by any effect. However, we have not yet discussed what to do when an effect's state changes only match part of the delta-state. In this case, the effect is split into two effects; the action-example is positive for one effect and negative or indeterminate for the other effect.

The positive and negative examples of the original effect are still positive and negative examples of the new effects. This means that preconditions of the original effect can be used to initialize the preconditions of the new effects.

```
procedure **Split-Effect**

Given:  op: An operator.
        eff: An effect of op.
        ex: An action-example of that operator.
        meff: State changes of eff that match ex.
        feff: State changes of eff that do not match ex.
Result: Split eff into two effects.

1. For operator op, create a new effect new-eff.
2. Copy the preconditions of the original effect eff to new-eff.

     s-rep(new-eff) ← s-rep(eff)
     h-rep(new-eff) ← h-rep(eff)
     g-rep(new-eff) ← g-rep(eff)

3. Copy the unused negative examples from eff to new-eff.
4. Set the state changes of the effects so that
     the action-example ex is a positive example of eff and
     a negative example of new-eff.

     state-changes(eff) ← meff
     state-changes(new-eff) ← feff

5. Refine eff with the action-example ex by invoking Refine-Positive-Example.
6. Refine new-eff with the action-example ex by invoking Refine-Negative-Example.
```

Figure 5.19: Splitting an Effect

The algorithm for splitting effects is shown in Figure 5.19 and illustrated with the data in Figure 5.20. In Figure 5.20, the action-example is a positive example of new effect 1 and a negative example of new effect 2. When new effect 1 is refined with the positive example, the h-rep and s-rep have one condition, (valve2 open), removed. When new effect 2 is refined with the negative example, the g-rep has one condition, (valve2 open), added.

The above discussion of splitting effects and reusing the original preconditions begs the question – why doesn't each effect's state change contain only one condition. This would remove the need to split effects. However, Diligent is an interactive system, and it takes less work for an instructor to examine and maintain one effect's preconditions than it would if several effects had duplicate preconditions.

Action-example:
- **Pre-state**:
  - (valve1 open)
  - (valve2 shut)
  - (HandleOn valve1)
  - (AlarmLight1 off)
- **Delta-state**:
  - (valve1 shut)

Original effect:
- **State changes**:
  - (valve1 shut)
  - (AlarmLight1 on)
- **Preconditions**:
  - **g-rep**:
    - (valve1 open)
  - **h-rep**:
    - (valve1 open)
    - *(valve2 open)*
  - **s-rep**:
    - (valve1 open)
    - *(valve2 open)*
    - (HandleOn valve1)
    - (AlarmLight1 off)

New effect 1:
- **State changes**:
  - (valve1 shut)
- Preconditions:
  - **g-rep**:
    - (valve1 open)
  - **h-rep**:
    - (valve1 open)
  - **s-rep**:
    - (valve1 open)
    - (HandleOn valve1)
    - (AlarmLight1 off)

New effect 2:
- **State changes**:
  - (AlarmLight1 on)
- **Preconditions**:
  - **g-rep**:
    - (valve1 open)
    - *(valve2 open)*
  - **h-rep**:
    - (valve1 open)
    - *(valve2 open)*
  - **s-rep**:
    - (valve1 open)
    - *(valve2 open)*
    - (HandleOn valve1)
    - (AlarmLight1 off)

Figure 5.20: An Example of Creating a New Effect

5.9 Complexity Analysis

This section analyzes the complexity of the learning algorithms.

Let

a	= number of attributes
	= maximum number of conditions in action-example pre-states, post-states and delta-states
c	= maximum length of space to represent a condition
i	= maximum length of an identifier that represents a condition (i should be a lot smaller than c)
v	= maximum number of values for each attribute
m	= maximum number of steps in a demonstration
t	= maximum number of action-examples for an operator
w	= maximum number of unused negative examples per effect
o	= maximum number of operators
e	= maximum number of effects in an operator

In the following, sets of conditions are represented as lists. The elements of these lists are ordered by attribute name. A list can contain at most one condition for any one attribute.

In order to avoid discussing the merits of different list implementations, the following discussion will make some assumptions. It is assumed that lists are implemented with pointers and that many list operations take O(1) time. These include deleting an element, appending an element to the end and inserting an element in the middle. Of course, finding where to delete an element or where to insert an element may require traversing the list and take O(a) time.

We will also assume that lists of action-examples are stored using identifiers and that copying them takes negligible time.

Comparing ordered lists of conditions. In the following, we will repeatedly compare two ordered lists of O(a) conditions in order to extract some elements from the lists or to merge the lists. This takes O(a) time. We will depend on the lists being ordered by attribute name. Consider finding the common conditions in two lists. The lists are compared by traversing them and comparing the current element in each list. If the elements are equal, a match is found, and the condition in one list can be appended to the list of matching conditions in (O(1)) time. If one element is less than the other, the lesser element is not in the other list. When a list's current element is found to be missing from the other list, the list's current element is changed to the list's next element. Since each list has at most O(a) elements, there are at most O(a) comparisons.

Action-Examples. We will first look at time complexity. The action-example needs to be created and the conditions ordered. The pre-state, post-state and delta-state each have O(a) conditions. The conditions need to be first copied (O(a)) and then sorted (O($a$log(a))). Thus, the complexity is O($a + a$log(a)) = O($a$log(a)).

Now look at space complexity. The pre-state, post-state and delta-state each have O(a) conditions. It takes O(c) space to represent each condition. There are O(t) action-examples for O(o) operators. A naive method is to store each condition with each action-example. In this case, the space required for all operators and action-examples is O($acto$). A better approach is to assign each condition a distinct identifier of length O(i) and use identifiers in action-examples. This takes O($aito$) space. Representing conditions by identifiers requires one identifier for each attribute value or O(acv) space. Thus, the space required for all action-examples is O($aito + acv$).

Creating Operators. Here we are only concerned about time complexity. The new operator has one effect. The effect's state changes and s-rep can have O(a) conditions. The g-rep is empty (O(1)). The h-rep can get O(a) conditions from the action-example's delta-state, but the attribute values in the conditions are incorrect. Thus, the conditions from the delta-state need to be checked against the action-example's pre-state with at most O(a) comparisons. The h-rep can also get conditions from the delta-state of action-examples for the demonstration's earlier steps. Since each earlier step provides at most O(a) conditions, merging lists for the m earlier steps takes O(ma). Thus, the time complexity for creating an operator is O(ma).

Comparing incompatible effects. Sometimes the preconditions of incompatible effects are compared.

We will look at time complexity. There are O(e) incompatible effects. For each incompatible effect, there are at most three comparisons between pairs of ordered lists of preconditions that contain O(a) attributes. Since the comparison with each list takes O(a), the time complexity is O(ae).

Processing a negative example. We will look at time complexity. A negative example's potentially needed conditions are compared to an effect's g-rep. There are O(a) attributes, and the comparison takes O(a). If a near-miss is found, one condition is inserted in the g-rep and h-rep. Inserting the condition requires O(a) comparisons to find to where to insert the condition. If a condition is not added, the current effect may be compared against incompatible effects, which takes O(ae)(see above). Thus, the time complexity is O(ae).

Processing a positive example. We will look at time complexity. The three precondition concepts (i.e. s-rep, h-rep and g-rep) are compared against an action-example's pre-state. There are O(a) attributes, and the comparison takes O(a). Afterwards, the O(w) unused negative examples are processed. Since processing a negative example takes O(ae), the processing of O(w) negative examples takes O(wae). Thus, the time complexity is O(wae).

Splitting an effect. We will look at time complexity. The O(a) attributes in the existing effect's s-rep, h-rep, g-rep and state changes are copied in O(a) time. Then one effect is refined with a negative example (O(ae)), and the other one is refined with a positive example (O(wae)). Thus, the time complexity is O(wae).

Creating a new effect. We will look at time complexity. The new effect's state changes come from the action-example's delta-state and contain O(a) conditions. The s-rep initially has O(a) conditions, and the g-rep is empty (O(1)). In the same manner as when the operator was created, some h-rep conditions are found in the action-example's delta-state (O(a)). Additional h-rep conditions are found using the h-rep of the operator's first effect (O(a) conditions). Comparing first effect's preconditions against the action-example's pre-state takes O(a) and then merging the conditions with the partial h-rep takes O(a). More h-rep conditions are come from differences between the pre-states of the action-example and the most similar negative example. Finding the negative example takes O(at) comparisons because it involves comparing O(a) attributes within O(t) action-examples. The differences between the action-examples are then merged with the partial h-rep in O(a) time. Thus, the time complexity is O(at)

Refining an operator with an action-example. Diligent is an interactive system and cannot spend too much time processing any one action-example. Therefore, we will look at the time complexity to update an operator with one action-example. First, the action-example needs to be created (O($a$log(a))). Second, the O(a) conditions in the action-example's delta-state are compared against the state changes of O(e) effects (O(ea) comparisons). Finally, the O(e) effects are refined with the action-example. Let R represent refining an operator with an action-example.

$$\begin{aligned}
\mathrm{O}(R) &= \mathrm{O}(\text{cost of creating action-example}) + \\
&\quad \mathrm{O}(\text{cost of comparing the action-example to each effect's delta-state}) + \\
&\quad \mathrm{O}(e(\text{cost of refining a positive example})) + \\
&\quad \mathrm{O}(e(\text{cost of refining a negative example})) + \\
&\quad \mathrm{O}(e(\text{cost of splitting a conditional effect})) + \\
&\quad \mathrm{O}(\text{cost of creating a new effect}) \\
&= \mathrm{O}(a\log(a)) + \mathrm{O}(ea) + \mathrm{O}(wae^2) + \mathrm{O}(ae^2) + \mathrm{O}(wae^2) + \mathrm{O}(at) \\
&= \mathrm{O}(a\log(a) + wae^2 + at)
\end{aligned}$$

5.9.1 Scalability

Diligent's approach is scalable because operators are learned for a particular object with relatively few action-examples. Because there are so few action-examples, it's reasonable to maximize learning by spending a little extra time on each action-example.

We will discuss the scalability issues from the previous section that appear most important. They are the space required to store action-examples, the time for creating new effects, the time for processing a positive example, and the time for splitting an effect.

The area for storing action-examples is greatly reduced by associating identifiers with conditions and storing the identifier rather than the condition in the action-example. The savings in space increases as more action-examples are created because most conditions appear in many action-examples. Furthermore, the same identifiers can be used in action-examples for all operators.

The space saved by using identifiers to represent conditions also enables the storage of action-examples in hash table. Storing action-examples in a hash table allows Diligent to check for duplicate action-examples before creating and storing a new action-example. This is important because Diligent tends to receive duplicate action-examples.

If the space required by action-examples becomes an issue, a limit could be placed on the number of previous action-examples stored.

Another scalability issue is the time it takes to create a new effect. Creating a new effect involves identifying h-rep conditions by comparing the the current positive example against the operator's previous action-examples. This is reasonable because the operator represents the manipulation of one object and has relatively few action-examples. If time became an issue, the number of previous action-examples examined could be limited.

A third scalability issue is the time to refine an effect with a positive example. The time spent on the positive example is not the issue. Instead, it's the time spent processing the unused negative examples. These negative examples differ from the s-rep in two or more conditions but are still classified as positive by the g-rep. However, processing these action-examples is not a problem because there tend to be only a few of them. Furthermore, the number action-examples doesn't get large because, as more negative examples are seen, the g-rep gets more refined and rejects more negative examples.

The final scalability issue is the time to split an existing effect in two. This has same time complexity as processing a positive example. Splitting an effect happens much less often than processing a positive example, and the time complexity of splitting an effect is dominated by time complexity of processing a positive example, which we have already discussed.

5.10 Related Work

Throughout this chapter, related work has been discussed where applicable. However, some other work should be mentioned.

Diligent can learn in an unstructured environment that does not have any explicit representation of the relationships between objects and attributes. Another algorithm for learning in this type of environment is MSDD [OC96], which learns probabilistic state changes. However, MSDD requires much more data than is available to Diligent.

Diligent is a Programming By Demonstration (PBD) system that focuses on determining which attributes are important. However, many PBD systems for manipulating graphical objects have a different type of environment. Instead, these systems have structured environments, which contain explicit relationships between objects and attributes. Learning in these systems tends to focus on identifying which relationships are important and generalizing object classes. An example of this type of system is Metamouse+ [MWM94].

Disciple has been used in a variety of domains [TK90, THD95, TH96, TK98]. Like Diligent, Disciple uses a version space algorithm with a single conjunctive concept for its upper and lower bounds (i.e. g-rep and s-rep). Unlike Diligent's g-rep and s-rep, Disciple's initial upper and lower bounds are heuristically altered so that they are only probable upper and lower bounds. These heuristic bounds define what is called a Plausible Version Space [Tec92]. To create these bounds, Disciple uses information about its structured environment that is unavailable to Diligent. Disciple overcomes errors in its bounds by allowing conditions to be added and removed from both the upper and lower bounds.

Like Diligent, a few PBD systems have used a version space algorithm for learning preconditions. Metamouse+ [MWM94] learns graphical editing procedures in an environment that is very different than Diligent's. Disciple [TH96], which is described above, has also been taught by demonstration. Recently, Lau and Weld [LW99] used an e-mail processing domain for comparing algorithms that learn preconditions. They looked at a version space and an inductive logic algorithm; however, their environment is very different than Diligent's, and their version space algorithm only learned a single precondition for an entire procedure.

Utgoff [Utg86] has looked at speeding up version space learning by dynamically creating attributes whose values are inferred from other attributes. This is inappropriate for Diligent because there is little data and because humans may not find the inferred attributes either understandable or reasonable.

5.11 Summary

This chapter discussed how Diligent learns operators. It focused on how Diligent identifies the preconditions necessary for an action to produce desired state changes. Good preconditions are important because Diligent uses them to derive a plan's step relationships.

First, we covered some requirements specific to learning operators. Diligent needs to quickly and incrementally learn operators with potentially little data. For these reasons, Diligent needs to be able to correct errors in the operators. Additionally, the operator representation needs to be usable by the human instructor. He needs to be able to understand the preconditions and to determine whether Diligent believes that a specific condition is a precondition. It would also be useful to provide him with some measure of confidence in a precondition. Finally, Diligent's environment contains many attributes, most of which are not needed by a given procedure. Thus, Diligent's learning methods need to identify attributes that are likely to be important.

Two types of data are provided to support learning: examples of actions being performed and the sequence of steps in the current demonstration.

Diligent processes the data using three heuristics. One heuristic assumes that attributes that changed value earlier in the demonstration are likely preconditions. This heuristic is used for creating new operators. The second heuristic favors existing knowledge. This means that Diligent should use what it already knows rather than general heuristics. This heuristic is used throughout the learning algorithm, but it particularly influences the creation of new effects when the operator already has an effect. The third heuristic favors extraneous preconditions over missing ones because it is easier to remove unnecessary preconditions than to add missing ones.

Preconditions are associated with effects, and an effect represents preconditions using a modified version space that has three sets of conjunctive conditions. The version space still has a most general bound (g-rep) and a most specific bound (s-rep), but Diligent augments the version space with an intermediate, best guess precondition (h-rep). The h-rep supports learning reasonable preconditions quickly and is used when calculating a plan's step relationships. The s-rep and g-rep are used for incremental learning, error recovery, and indicating Diligent's confidence in a particular precondition. If Diligent is very confident, the precondition is in the g-rep, and if Diligent strongly believes a precondition is unnecessary, then the precondition is not even contained in the s-rep.

We also discussed how Diligent refines preconditions using action-examples. Positive examples remove unnecessary conditions from the s-rep and h-rep, while negative examples add conditions to the h-rep and g-rep.

Finally, we looked at the algorithm's complexity and argued that the approach is scalable.

Chapter 6

Experimenting

In the previous chapter, we discussed learning operators. Operators are associated with each step in a procedure and identify the step's preconditions and state changes. Diligent uses these preconditions and state changes to derive the dependencies (i.e. step relationships) between steps.

Procedures containing these dependencies will be used by an automated tutor to teach human students. Consequently, errors in the dependencies may mislead students.

One source of errors is the lack of training data. Because the instructor has limited time, Diligent may only see a step demonstrated a few times. This forces Diligent to use heuristics when creating preconditions. Unfortunately, heuristic preconditions may contain mistakes, and the quality of the preconditions determines the likelihood of errors in a procedure's step relationships.

One method for refining preconditions is to perform a step in several different states and observe what happens. Diligent does this when it performs experiments.

Besides performing steps in multiple states, experiments need to meet a variety of objectives. They should minimize the work performed by the instructor. They should exploit Diligent's access to the environment and focus attention on the procedure being learned. Experiments should also compensate for the bias in the heuristics used for creating preconditions.

Diligent meets these objectives with a novel technique: Diligent performs steps in a variety of states during autonomous experiments that are generated from demonstrations of a procedure. Demonstrations are useful because they specify a sequence of steps that can be used to perform a procedure. When experimenting, Diligent performs the procedure but skips a step. Diligent then observes how skipping the step affects subsequent steps. Since the heuristics used for creating preconditions assume that the state changes of earlier steps are likely preconditions for later steps, skipping steps helps compensate for the heuristic bias.

This chapter discusses using experiments to refine the preconditions of operators. First, we will describe the problem in terms of requirements. Afterwards, we will discuss issues that motivate Diligent's approach. We will then discuss Diligent's approach. Finally, we will end with a discussion of the run-time complexity and related work.

6.1 The Problem

Earlier in Section 3.1, we described the authoring problem in terms of requirements, constraints and the interface to the environment. Because the problem has become more constrained and concrete, we will define some new requirements.

6.1.1 Requirements

Let us consider how the requirements in Chapter 3 relate to experiments. The experimentation approach needs to help understand demonstrations by getting the most out of each demonstration. Experiments should save the instructor time and reduce the difficulty of authoring. When experimenting, Diligent should exploit its ability to access and manipulate the environment.

We will also define the following additional requirements.

Generate more examples of steps being performed. The goal of experimentation is to better understand the dependencies between a procedure's steps. To do this, the operator learning algorithms need examples of the steps being performed in a variety of states so that operator preconditions can be refined.

Compensate for operator learning bias. Some errors in operator preconditions may result from the bias that favors attributes that change value during a demonstration. The bias is reasonable because changes caused by earlier steps are often preconditions for later steps. However, some of these preconditions may be incorrect.

Positive and negative examples should be similar. A *positive example* is when an action produces the desired state changes, and a *negative example* is an example that is not positive. Positive examples help eliminate unnecessary preconditions, while negative examples identify necessary preconditions.

In Diligent's learning algorithm, it is harder to process negative examples than positive ones. Unlike positive examples, a negative example requires a near-miss (or one condition) difference between its pre-state and the most specific candidate precondition (i.e. s-rep). The likelihood of finding a near-miss increases when negative and positive examples are similar.

Interactive system should be fast. Diligent is an interactive system for instructors with a limited amount of time. If experiments force instructors wait long periods of time or even stop, then instructors may have difficulties because a loss of concentration and focus. Thus, general purpose techniques that do not focus on understanding the procedure could take a prohibitive amount of time.

A related concern is why are Diligent's experiments performed interactively when they could have been done off-line. The reason is that Diligent's experiments focus on understanding demonstrations and interactive experiments make the tool easier to use. If experiments were performed off-line, an instructor might have to wait a long time to see what an experiment learned. In contrast, an instructor can quickly see the results of interactive experiments. Because an instructor waits less time, it should be easier for him to concentrate and focus on the procedures being authored.

Bounded number of steps in an experiment The time required to perform experiments can be controlled only if a limited number of steps are performed. While it is reasonable to perform additional steps in response to an unexpected observation, the number of additional steps should not be too large or unpredictable.

The requirements for being fast and bounding the number of steps argue against using autonomous discovery algorithms that may perform a large, unpredictable number of steps. This also argues against using techniques that may require a bounded but large number of steps. This includes systems that attempt to build a correct finite state automaton of the environment [Ang87b, Ang87a, RS90, She94].

6.2 Background

This section discusses issues relevant to Diligent's experimentation approach. We will also discuss other approaches that are inappropriate for Diligent, but might complement Diligent's approach in a future system.

6.2.1 Focused versus Unfocused

An important issue is why a system performs experiments. Diligent experiments because it's attempting to understand a given procedure well. Experiments that concentrate on gaining general knowledge may learn to do a lot of things well, but are likely to take more time and be less useful than those that focus on understanding the steps of the given procedure.

6.2.2 Supervised versus Unsupervised

Experiments can be viewed as generating a question, asking an "oracle" the question, and waiting for the oracle to provide the correct answer. When the oracle is human, the experiment is *supervised*, and when the oracle is automated (e.g. Diligent's environment), the experiment is *unsupervised*.

Some systems perform supervised experiments by generating potential examples of a concept and then asking the user whether they are examples of the concept. Humans often find this type of yes or no question easy to answer. Systems that use structural domain knowledge (i.e. class hierarchies and relations between objects) for generating examples include ALVIN [KW88], MARVIN [SB86] and Disciple [TK98, TH96].

This approach is inappropriate for Diligent because Diligent solves a different problem. Not only does Diligent try to minimize the effort required by the instructor, but Diligent's unstructured environment does not provide class hierarchies or relations between objects.

Still, Diligent could be viewed as performing supervised experiments when it asks the instructor to verify goal conditions, ordering constraints and causal links.[1] However, these questions verify information after it has been computed rather than providing input for machine learning algorithms.

In contrast to supervised experiments, unsupervised experiments reduce the instructor's work by letting the environment answer the questions. Unsupervised experiments also reduce the possibility of instructor error. Systems that perform unsupervised experiments include EXPO [Gil92], OBSERVER [Wan96b] and LIVE [She93]. The method used by these systems to experiment will be discussed in the next section.

[1] See Chapter 4.

6.2.3 Experimenting with Plans

Some systems experiment by building plans that transform an initial state into a goal state. This can be done two ways: practice problems and explicit experiments. A *practice problem* requires a system to create a plan that transforms an initial state into a goal state. An explicit experiment has two components: an action to be performed and a desired state in which to perform the action. Placing the environment into the desired state often involves solving a practice problem where the current state is transformed into the desired state. Both practice problems and explicit experiments allow a system to learn by observing how various actions affect the environment. Systems that learn by creating and performing plans include LEX [MUB83], LIVE [She93, She94], OBSERVER [Wan96c], EXPO [Gil92, CG90] and IMPROV [Pea96].

When creating plans several issues need to be addressed.

- What knowledge does a system utilize when creating a plan? OBSERVER only utilizes knowledge of operators, and it learns by deliberately not satisfying some potential preconditions. In contrast, EXPO uses sophisticated domain independent techniques. EXPO identifies missing preconditions by favoring hypothesized preconditions that involve 1) attributes of objects involved in the action, 2) predicates that appear in all successful past situations, and 3) operators similar to the one being examined. EXPO also restricts the space of plans with seventeen rules that favor certain types of plans (e.g. avoid long plans).

- When is an experiment finished? Does it require the goal state to be reached or does it merely involve performing the steps? Attempting to reach the goal after the initial plan fails could take a large number of steps.

- How are practice problems generated? Are they automatically generated as in EXPO, or does someone need to generate them as in OBSERVER? An advantage of automatically generating problems is that the system has some control over a problem's apparent difficulty, while an advantage of user selected problems is that the user can guide learning.

Another issue is whether the system is doing an extensive search during planning or whether it is doing more limited and controlled planning. An extensive search could take a long time, while more limited planning might just involve small changes to an existing plan.

For Diligent, an extensive search is inappropriate. If the system were busy experimenting, the instructor could not provide additional demonstrations. Furthermore, if each demonstration resulted in a long delay, instructors might be hesitant to provide additional demonstrations. Recall that one of Diligent's objectives is allowing instructors to easily author procedures with demonstrations.

Diligent's potentially limited knowledge is more compatible with limited planning. However, limited planning raises issues beyond Diligent's scope. Diligent would need to identify which plans to create and when they should be created. This may be difficult when Diligent has seen only a few demonstrations and knows only a few poorly understood operators. Depending on how plans are created, Diligent might not yet have the minimum knowledge necessary for planning.

Instead, Diligent needs a basic approach that can be reliably used even when the system has very minimal knowledge. At this stage, Diligent could have difficulty solving practice problems whose solutions differ only slightly from a demonstration. That is why the more knowledge-intensive planning techniques of EXPO and OBSERVER are not used.

Nevertheless, planning could complement Diligent's experimentation approach. Diligent's experiments could be used as an initial phase to refine operators and to identify situations that merit the use of planning. Later, when enough knowledge has been built-up, more planning intensive techniques could be used.

6.3 Input

This section describes the input used for performing experiments and defines terms that should make the following discussion easier to understand.

Diligent experiments on procedures, whose basic structure was described in Section 4.3.

Procedures contain one or more paths. A *path* describes an initial state and a sequence of steps. The sequence of steps in each path is specified by one or more demonstrations. A path can represent multiple demonstrations because a demonstration can add steps to an existing path. Diligent actually uses paths rather demonstrations to generate experiments.

The specification of a path's initial state is called a *prefix*. A prefix identifies a known configuration of the environment (Section 3.1.3) and a sequence of actions that alters the configuration.

A *step* represents a portion of the procedure. Most steps are primitive. A *primitive* step represents an action performed in the environment. If a step is not primitive, it is abstract. An *abstract* step represents a subprocedure that contains its own steps. A procedure containing an abstract step is called *hierarchical*.

When Diligent performs an abstract step, it attempts to establish the goal conditions of the abstract step's subprocedure. A *goal condition* indicates the value an attribute should have when the subprocedure is finished. To establish the goal conditions, some of the subprocedure's steps are performed.

Associated with each primitive step is an operator (Section 3.2.2.2). An *operator* represents an action performed in the environment and identifies the preconditions needed to produce given state changes. Because actions can produce different state changes when performed in different states, some of an operator's state changes may have different preconditions. The purpose of experiments is to refine the preconditions of operators.

The preconditions of operators are refined with *action-examples*, which contain the state of the environment before (*pre-state*) and after (*post-state*) performing an operator's action.

6.4 Diligent's Approach

Diligent experiments by repeatedly performing a procedure but altering it so that a different step is skipped each time. Before performing the procedure, Diligent uses its ability for resetting the environment so that it, like a student learning the procedure, starts the procedure from a specified initial state. As Diligent performs the procedure, it observes how skipping the step affects later steps. This examination of how the state changes of earlier steps affect later steps helps compensate for bias used when creating operators. When all steps have been performed, the experiment is finished. The experiment is finished because its purpose is generating action-examples of the procedure's steps rather than achieving some goal state. This approach should be quick because it bounds the number of steps performed in an experiment.

Because a procedure's steps are specified by demonstrations, experiments are really generated from demonstrations. Generating unsupervised experiments from demonstrations serves a number of purposes. It doesn't require accurate domain knowledge. It addresses Diligent's requirements to understand demonstrations and to make the instructor's job easier by making more use of each demonstration. It also uses Diligent's heuristic focus on attributes that change value, and it exploits Diligent's ability to interact with the environment, which includes the ability to reset the environment's state and to perform actions.

The approach focuses on validating the preconditions created by the heuristics used to create operators. One heuristic is that attributes that changed value earlier in a demonstration are likely to be preconditions of later steps. This results in a bias towards creating unnecessary preconditions. Experiments remove these unnecessary preconditions when they show that a later step is not dependent on an earlier step. Experiments may also identify when a later step is dependent on an earlier step. When this happens, experiments not only provide evidence that preconditions are correct but also support the identification of missing preconditions.

As mentioned earlier, positive examples remove preconditions, and negative examples add and verify preconditions. Because experiments focus so closely on the procedure, positive and negative examples tend to be similar. This similarity should be beneficial because there may be few action-examples and using a negative example requires a one condition difference between it and potential preconditions.

This approach is straightforward for primitive steps, but how does it handle steps that represent subprocedures? In this case, Diligent uses an heuristic that focuses on the current procedure. This means that, as much as possible, abstract steps (i.e. subprocedures) should be treated like other steps. In other words, an abstract step is treated as black box that achieves the goal conditions of its subprocedure. To allow a subprocedure to achieve its goal conditions, Diligent internally simulates performing the subprocedure in order to determine which of the subprocedure's steps to perform. Of course, when performing an experiment, an abstract step, like other steps, may sometimes fail to establish the desired state changes.

Diligent's focus on the current procedure reduces the number of steps in an experiment. Because there are fewer steps, the instructor doesn't have to wait as long.

6.5 The Procedure Being Used

As a procedure, we will use the extended example from the chapter on processing demonstrations (Chapter 4). Figure 6.1 shows the extended example. The steps representing a procedure's beginning and end (e.g. begin-proc1 and end-proc1) are not shown because the experimentation algorithm ignores those steps. The procedure that we will experiment on is top-level, which uses procedures proc1 and proc2 as subprocedures.

The steps and procedures do the following. Procedure top-level shuts two valves and checks an alarm light. Procedure proc1 shuts two valves, and procedure proc2 checks an

Procedure top-level:
Steps: turn-5 → proc1-6 → proc2-7

Procedure proc1
Steps: turn-1 → move-2nd-2 → turn-3 → move-1st-4

Procedure proc2
Steps: press-test-8 → check-light-9 → press-reset-10

Figure 6.1: A Hierarchical Procedure

alarm light while in test mode. In our later discussion, we will use the fact that steps turn-5 and turn-1 both shut valve1.

6.6 The Algorithm

```
procedure Experiment-On-Procedure

Given:    proc : a procedure.
Result:   Perform experiments the procedure's paths.

1. Initialize the stack of experimental commands expr-stack as empty.
2. For each path pth of the procedure do the following.
       3. If path pth has not been updated since it was in an experiment, then
          generate experiments for pth and add them to expr-stack.
          Do this with Gen-Skip-Step-Experiment.
4. Perform the experiments contained in expr-stack using Perform-Experiment.
```

Figure 6.2: The Top Level Experimentation Algorithm

Diligent performs experiments using procedure **Experiment-On-Procedure** (Figure 6.2). On line 1, the stack of experimental actions to perform is emptied; this merely puts the stack into a known state. On lines 2 – 3, experiments are generated for each of the procedure's paths. The experiments are stored in the stack *expr-stack*. Afterwards, on line 4, experiments are actually performed.

The experiments are generated by procedure **Gen-Skip-Step-Experiment** (Figure 6.3). In an experiment, the path's initial state is reset and all but one of the procedure's steps are performed. This is done for every step but last step in the path. Two types

```
procedure Gen-Skip-Step-Experiment

Given:   proc: A procedure.
         pth: A path with $n$ steps.
         expr-stack: A stack of experimental commands to perform.
Result:  expr-stack: Updated stack of commands.

1. Loop over $i$ where $i$ goes from 1 to ($n$ - 1)
        2. Compute the sequence seq of steps to perform;
           include all the path's steps except the $i$th step.
           (If the path has steps $s_1 \ldots s_n$, then $seq = s_1 \ldots s_{i-1} s_{i+1} \ldots s_n$.)
        3. Push each of seq's steps onto expr-stack as perform-step
           commands. Start with the last step in seq and work backwards to
           the first step. (Pushing the steps in reverse order causes the path's
           earlier steps to be performed before the path's later steps.)
        4. Push the path pth's prefix onto expr-stack as an
           reset-environment command. (The command will be used to
           reset the path's initial state.)
```

Figure 6.3: Generating Skip-Step Experiments

of commands are placed in the stack of experimental commands: perform-step and reset-environment. A *perform-step* command performs one of the path's steps, and a *reset-environment* command resets the environment's state to path's initial state. Notice that a path's steps are pushed onto the stack in reverse order so that a path's later steps are performed after its earlier steps.

The stack of experimental commands looks like a) of Figure 6.5 after **Gen-Skip-Step-Experiment** has processed procedure top-level, which has only one path. The stack indicates that the procedure will be performed twice: once skipping the first step (turn-5) and once skipping the second step (proc1-6). As we discussed before, abstract steps proc1-6 and proc2-7 (i.e. subprocedures proc1 and proc2) are treated the same as primitive step turn-1.

The procedure **Perform-Experiment** is shown in Figure 6.4. Until the stack (*exper-stack*) is empty, the procedure keeps popping off and processing the top command in the stack (lines 1 and 2). When **perform-experiment** is invoked, the stack looks like a) in Figure 6.5, and when it finishes, the stack is empty.

The procedure **Perform-Experiment** first processes a reset-environment command (line 4 of Figure 6.4). Performing this command restores the path's initial state.

```
procedure **Perform-Experiment**

Given:   *exper-stack*: Stack of experimental commands to perform.
Result:  Perform all the commands in *exper-stack*.

1. While *expr-stack* is not empty
     2. Pop the top command off of *expr-stack*.
     3. Based on the type of command, do one of the following:

        4. If the command is a reset-environment command, then
           restore the path's initial state using **Replay-Prefix** (Section 4.6.6.1)
           and the prefix associated with the command.

        5. If the command is a step-perform command and the step is
           primitive, do one of the following:
             6. If the step just senses the environment without changing it
                (i.e. a sensing action), do nothing.
             7. Otherwise, perform the step's action. This is done with the
                action-id of the step's operator and **Perform-Action**
                (Section 3.1.3). This produces an action-example that is used
                to update the step's operator with
                **Refine-Operator** (Section 5.8.1).

        8. If the command is a step-perform command and the step is
           abstract (i.e. a subprocedure), then
             9. Compute the sequence *seq* of steps needed to perform
                the subprocedure from the current state with
                **Internally-Simulate-Subprocedure** (Section 4.7.1).
             10. Push each step in *seq* onto *expr-stack* as
                a perform-step command. Start with the last step in *seq* and
                work backwards to the first step. By working backwards,
                steps that are earlier in *seq* will performed before later steps.
```

Figure 6.4: Performing Experiments

a) After skip-step experiments have been generated

reset → proc1-6 → proc2-7
  → reset → turn-5 → proc2-7

b) Before processing step proc1-6

proc1-6
  → proc2-7
  → reset → turn-5 → proc2-7

c) After processing step proc1-6

turn-1 → move-2nd-2 → turn-3 → move-1st-4
  → proc2-7
  → reset → turn-5 → proc2-7

Figure 6.5: The Stack of Actions to Perform

If the type of command is perform-step and the associated step is primitive, then Diligent performs the step's action in order to refine the action's operator (lines 5 – 7). Line 6 deals with sensing actions, which are actions that gather knowledge from the environment without changing it (e.g. check whether a light is illuminated). Diligent assumes that the environment allows sensing actions to be performed successfully in any state.

Because sensing actions do not change the environment and can be performed successfully in any state, it is unclear whether Diligent can learn anything from them. Instead of changing the environment, sensing actions create mental attributes that record the current values of environment attributes. However, Diligent only checks for the existence of mental attributes and does not consider their values. Given that mental attribute values are ignored, Diligent could only potentially refine a sensing action's preconditions.[2] Consider a procedure (e.g. proc2) that checks the state of a light while the system is in test mode; outside of test mode, it is irrelevant whether the light is on or off. How could a system with limited knowledge, such as Diligent, know that being in test mode is mandatory? For this reason, Diligent ignores sensing actions during experiments.

If a future system used the values of mental attributes, then performing sensing actions during experiments might be useful. This is area for future work.

[2]A sensing action's preconditions are used to control when the sensing action's step is performed. For this reason, a sensing action's hypothesized preconditions are associated with the sensing action's step rather than its operator. The assumption is that a sensing action's preconditions are specific to the given step and procedure rather than independent of a procedure like the preconditions of operators.

If the type of command is perform-step and the associated step is abstract, then Diligent treats the step as a black box that achieves the goal conditions of the step's subprocedure. Diligent does this by simulating the subprocedure (line 9). Simulating a subprocedure involves looking at the current state and determining which of the subprocedure's steps need to be performed. This means that Diligent may perform steps in the subprocedure that it would normally skip or skip steps that it would normally perform. Consider b) and c) in Figure 6.5. In b), the top command in the experimental stack is the abstract step proc1-6, which performs the subprocedure proc1. In c), step proc1-6 has been replaced by the steps of procedure proc1. Normally, when performing proc1-6, proc1's first step (turn-1) is skipped because step turn-5 has already shut valve1. However, during this experiment, step turn-5 is skipped. Because Diligent attempts to achieve the goal conditions of proc1, proc1's first step (turn-1) is performed.

6.6.1 What Was Learned From the Experiment

As mentioned earlier, the purpose of experiments is to refine operator preconditions. Therefore, we will briefly review how operators are represented. In an operator, each state change is associated with three conjunctive sets of preconditions. The most general set of preconditions, *g-rep*, contains conditions that have been shown to be necessary. The best guess set of preconditions, *h-rep*, contains likely preconditions, and the most specific set of preconditions, *s-rep*, contains unlikely preconditions. All conditions in the g-rep are contained in the h-rep and s-rep, and all conditions in the h-rep are contained in the s-rep.

Changes to the three sets of preconditions impact the procedure differently. It is desirable to remove conditions that are only in the s-rep, but the s-rep is not used when deriving a plan's step relationships. In contrast, changes to the h-rep or g-rep are important because Diligent uses them to derive step relationships.

We will now discuss what is learned when experimenting on the above procedures.

The above experiments illustrate how experiments are performed on hierarchical procedures. Experiments on hierarchical procedures focus on the current procedure and assume that subprocedures are already refined. However, experiments on a hierarchical procedure can refine subprocedures by performing them in different initial states. The experiments can also reveal unexpected dependencies between subprocedures.[3]

In our example, Diligent learned little when experimenting on the hierarchical procedure top-level. That is because top-level's three steps are relatively independent of each

[3]Extensions that deal with unexpected behavior in subprocedures will be discussed in Chapter 8.

other. Additionally, few steps were performed in new pre-states. The first step turn-5 is not needed by the second step proc1-6 because procedure proc1 contains step turn-1 that is equivalent to turn-5. The steps of subprocedure proc2 had some s-rep preconditions removed when step proc1-6 was skipped, but Diligent doesn't use s-rep preconditions when building plans.

If the instructor had experimented on procedure proc2, nothing would be learned because the procedure has only three steps and one of them represents a sensing action. Remember that sensing actions are ignored during experiments.

Step	**Old Preconditions**	**New Preconditions**	**State changes**
turn-1	(valve1 open)	*(valve1 open)* *(HandleOn valve1)*	(valve1 shut)
move-2nd-2	(valve1 shut) (HandleOn valve1)	(HandleOn valve1)	(HandleOn valve2)
turn-3	(valve1 shut) (valve2 open) (HandleOn valve2)	(valve2 open) *(HandleOn valve2)*	(valve2 shut)
move-1st-4	(HandleOn valve2)	(HandleOn valve2)	(HandleOn valve1)

The preconditions in *italics* have been identified as necessary.

Table 6.1: Changes to proc1's Preconditions

In contrast, experimenting on procedure proc1 would have updated preconditions and caused Diligent to derive different step relationships. The changes to the preconditions are shown in table 6.1. The preconditions shown italics are in the g-rep, while the others are in the h-rep. Notice that the preconditions are much better after the experiments.

However, the final preconditions in table 6.1 are not perfect. The steps move-2nd-2 and move-1st-4 should have no preconditions. Diligent is unable to remove the attribute for the pre-state location of the handle (i.e. HandleOn) because it has only seen the handle move between the two valves. The error would have been corrected if the instructor had demonstrated moving the handle from other valves. In any case, this is a subtle error that might escape the notice of an instructor and human students.[4]

One potential concern is that the upper and lower bounds of the version space (i.e. g-rep and s-rep) have not converged to the same concept. However, this convergence is highly unlikely given a potentially large number of attributes and the few action-examples.

[4]When evaluating Diligent (Chapter 7), none of the test subjects appeared to spot this error.

Even OBSERVER [Wan96c], which had a lot more data than Diligent, did not expect convergence. Nevertheless, the version space's upper and lower bounds are still useful because they provide the instructor with a measure of Diligent's uncertainty. Besides, Diligent's objective is to provide the instructor with an h-rep containing reasonable preconditions.

6.7 Complexity Analysis

This section discusses the run-time complexity of the experimentation algorithm.

In the following,

- We will consider a procedure to have one path.
- We will ignore the cost of resetting the environment's state and instead focus on the path's primitive steps. (The environment is reset before performing the procedure's steps.)
- Because sensing actions are not performed during experiments, we will not consider them.

Consider a one level procedure (i.e. without subprocedures). If the procedure has n steps, the procedure is performed $(n-1)$ times while skipping steps. Each performance of the procedure takes $(n-1)$ steps. Thus, experiments on a one level procedure perform $O(n^2)$ steps.

Unfortunately, when experimenting on a one level procedure, $\frac{1}{2}$ of the steps may not provide any information. The problem is that the steps before the skipped step merely perform the procedure. However, performing the procedure once might be useful if the procedure's path was created from multiple demonstrations because the path's steps may not have been performed sequentially from start to finish.

Experiments could avoid these unnecessary steps if the environment's state before the last skipped step could be quickly saved and restored. This capability would allow each performance of the procedure to start at a later step and a different initial state.

When hierarchical procedures are considered, the time complexity improves.

A procedure can be viewed as a tree, where the procedure is the root node and each primitive step is a leaf node. The *direct descendents* of a procedure are its primitive and abstract steps, while its *descendents* are all the nodes in tree whose root is the procedure. The length of the path from the root to a node is called its *depth.* The direct descendents of the root node have a depth of 1. The *height* of a tree is the maximum depth of any node. A procedure containing only primitive steps has a height of 1. A procedure of height 2 contains abstract steps, but the procedures performed by the abstract steps contain only primitive steps.

Let all procedures contain at most b direct descendents. Because a tree of a given height contains more nodes when it is balanced, we will assume that all procedures have b direct descendents and that a procedure's direct descendents are either all primitive or all abstract.

An upper bound on the number of leaves of a tree of height h is b^h [CLR90]. In other words, a procedure of height h contains at most b^h primitive steps.

Consider an experiment performed on the hierarchical procedure at the root node with height h. Diligent only experiments on a given procedure's direct descendents. Assume that the number of steps performed by subprocedures does not change because earlier steps were skipped. In this case, Diligent performs the procedure $(b-1)$ times while skipping a step. Each performance of the procedure uses $(b-1)$ direct descendent steps. Because each direct descendents is an abstract step, each of the direct descendents is realized by b^{h-1} primitive steps. Thus, total number of primitive steps performed in an experiment on the root procedure is at most

$$\begin{aligned} &= b^{h-1}(b-1)^2 \\ &= b^{h+1} - 2b^h + b^{h-1} \qquad \text{(X)} \end{aligned}$$

Now consider the case where the root procedure and every descendent subprocedure have experiments performed on them. We will prove by mathematical induction that this involves performing $h(b^{h+1} - 2b^h + b^{h-1})$ primitive steps, where $h \geq 1$.

Let $\mathbf{G(h)} \equiv h(b^{h+1} - 2b^h + b^{h-1})$.

Consider a procedure with only primitive steps (i.e. $h = 1$). In this case, $(b-1)$ of the procedure's steps are performed $(b-1)$ times. Because $G(1) = (b-1)^2$, the $G(h)$ holds for $h = 1$.

Now assume that $G(h)$ is correct for procedures of height $h-1$. Consider a root procedure of height h.

Each of the root procedure's direct descendents represents a procedure of height $h-1$. Since there are b direct descendents, the number of steps performed while experimenting on procedures other than the root procedure are

$$\begin{aligned} bG(h-1) &= b(h-1)(b^h - 2b^{h-1} + b^{h-2}) \\ &= (h-1)(b^{h+1} - 2b^h + b^{h-1}) \\ &= G(h) - (b^{h+1} - 2b^h + b^{h-1}) \qquad \text{(Y)} \end{aligned}$$

From (X), we know that the number of primitive steps performed while experimenting on only the root procedure is $b^{h+1} - 2b^h + b^{h-1}$. Now if we combine the steps performed for the root procedure with steps performed for the subprocedures (Y), we get a total of $G(h)$ primitive steps for performing all experiments.

Thus, by mathematical induction it has been shown that $G(h)$ bounds the number of primitive actions performed during experiments on a multi-level procedure of height h.

6.7.1 Scalability

Diligent's techniques are meant to be used with short procedures that can be combined into modular, hierarchical procedures. If procedures are authored in a hierarchical manner, the number of primitive steps performed by experiments decreases rapidly.

Consider a procedure containing 125 primitive steps. If the procedure were authored without subprocedures, experiments would perform over 15,000 steps. However, the same procedure could be authored in a hierarchical manner with a height of 3 and a branching factor of 5. In this case, experiments would only perform 1,200 steps.

However, we have never seen any procedures close to this length. In the two domains that we've looked at, the HPAC has the longest procedures. If we ignore sensing actions, almost all HPAC procedures take less than about 15 steps. The longest procedure appears to be about 45 steps, but most very long procedures use common subsequences of steps, such as checking all 14 temperature sensors or opening and closing all 5 separator drain manifold valves. These common subsequences could easily be modeled by reusable subprocedures.

Furthermore, our experience has been that the 1 or 2 minutes spent experimenting is a small portion of the authoring process.

Experiments on the hierarchical procedure may not learn as much about the procedure as experiments on a one level procedure, but Diligent's focus is not autonomous exploration. Instead, Diligent's experiments should provide a bounded, heuristic aid for identifying operator preconditions.

Although using a hierarchy of procedures helps, Diligent's approach to experimentation is probably inappropriate for very large procedures. As Diligent gains more experience, experiments are likely provide little additional knowledge because by then both operators and subprocedures are likely to be very refined. Instead, Diligent's approach appears more appropriate for small subprocedures that can be used to construct large procedures.

6.8 Related Work

When appropriate, related work has been mentioned throughout this chapter. However, some other work should be mentioned.

6.8.1 The Self-Explanation Effect

The self-explanation effect [CBL$^+$89, CV91, CLCL94, Ren97] describes the phenomenon where human students can solve procedural problems better if they study a few problem solutions in detail rather than many solutions briefly. The term “self-explanation” is used because students need to make a conscious and deliberate effort to justify each of the solution’s steps. Besides better problem solving, Chi et al. [CBL$^+$89] found that students who produced self-explanations when studying physics had a better understanding about gaps in their knowledge.

Although Diligent does not model human cognition, the self-explanation effect motivates Diligent’s experimentation technique of examining each demonstration in detail. Diligent’s demonstrations are comparable to the problem solutions given human students. To explain a demonstration, Diligent tries to understand how state changes caused by earlier steps affect later steps.

The self-explanation effect is modeled by CASCADE [VJC92, Van99], which models human students learning to solve physics problems by studying the solutions of problems. Instead of experimenting with a simulation like Diligent, CASCADE uses knowledge of domain theorems (e.g. physics laws) and problem modeling concepts. CASCADE has been used as the basis for acquiring knowledge for an automated tutoring system [GCV98]. If a knowledge acquisition system has easy access to a well-defined domain theory, then CASCADE’s approach might be appropriate. Unlike CASCADE, Diligent does not require direct access to a well-defined domain theory.

6.8.2 Other Systems

A system that experiments by systematically analyzing demonstrations is PET [PK86]. Unlike Diligent, PET has complete control of the state. PET attempts to understand a sequence of actions by systematically changing the state and then performing actions. However, Diligent cannot use this approach because Diligent has limited control over the environment’s state.

A system uses that uses demonstrations for generating experiments is CAP [HS91]. CAP observes another agent and creates a theory to describe a sequence of actions. Unlike Diligent, CAP uses inverse resolution to create new concepts and to generalize the object classes. As discussed in Chapter 5, Diligent solves a different learning problem. Furthermore, CAP reactively experiments when the environment is in an opportunistic state rather than systematically resetting the state and performing experiments.

If you consider a successful plan as equivalent to a demonstration, then some case-based systems can also use demonstrations to generate experiments. For example, CHEF [Ham89] performs experiments by adapting and repairing plans for Szechwan cooking. CHEF experiments by creating a plan and then getting feedback about plan failure from a simulation. The feedback consists of faults and reasons. A fault is an undesired attribute value, and a reason is a causal explanation for the fault. Instead of repairing plans, Diligent learns the type of causal knowledge that is returned to CHEF by the simulation.

6.9 Summary

In this chapter we discussed how Diligent performs autonomous experiments to help it understand the preconditions of a procedure's steps.

We first looked at how this problem fits into the general requirements: experiments should make the instructor's job easier while maximizing the use of the limited number of demonstrations.

We also discussed some specific requirements. Experiments should generate action-examples for refining the operators associated with the procedure's steps. The action-examples should help compensate for the bias used in creating operator preconditions. To promote learning, a step's action-examples should have similar pre-states so that positive and negative examples have similar pre-states. Because the system is interactive, it should be fast and should attempt to bound the number of steps in an experiment.

We then discussed why other approaches were inappropriate: they perform too many steps, require too much domain knowledge, require too much interaction with the instructor, and do not focus on understanding the demonstrations of the given procedure.

We then discussed Diligent's approach to experimentation. Diligent performs the procedure while skipping a step and observing how this impacts later steps. Diligent's approach does not require interaction with the instructor and focuses on understanding the given procedure's demonstrations. Furthermore, because Diligent does not attempt to achieve any goal state, each experiment has a bounded number of steps. The number of steps is further limited because experiments treat abstract steps the same as primitive steps. In other words, Diligent skips steps in the current procedure, but does not perform similar experiments in the subprocedures associated with abstract steps.

We finished by showing that hierarchical composition of larger procedures from smaller procedures can greatly reduce the number of steps performed during experiments.

Chapter 7

Empirical Evaluation

So far, we've discussed how Diligent understands demonstrations and how Diligent can be used for authoring. But is Diligent an effective tool for authoring? This chapter addresses this question. Specifically, a study was conducted where people authored procedures with different versions of Diligent. In the study, everyone authored the same procedures, but each subject only used a single version of Diligent. The different versions were then compared using variables such as accuracy, effort, time and subjective evaluation.

This chapter is organized as follows. First, we discuss the testable hypotheses and the three versions of Diligent that were used to test the hypotheses. We then discuss how we tested the usability of Diligent and its tutorial materials. Afterwards, we discuss the experimental method, the experimental results, and how the results support the hypotheses.

7.1 Hypotheses

In the evaluation, we were concerned about two hypotheses that dealt with the benefits of demonstrations and of experiments. One hypothesis is that demonstrations are beneficial even if Diligent does not perform experiments. To test this hypothesis, we compared subjects who used demonstrations without experiments against subjects who only used an editor. The other hypothesis is that using both experiments and demonstrations is better than using only demonstrations. To test this hypothesis, we compared subjects who used both demonstrations and experiments against subjects who only used demonstrations.

When testing these hypotheses, all subjects could use an editor. The subjects who only used an editor differed from the others in that they had to specify a procedure's steps with the editor. In contrast, the other subjects had to specify steps with demonstrations.

To measure these hypotheses, a number of testable claims were created. Each claim corresponds to one of the dependent variables.[1]

Claim 1: Subjects require less work to create a procedure when using demonstrations and experiments than when using only demonstrations. Work in this case means deliberative changes to Diligent's knowledge base rather than time spent authoring. For example, adding a step is a deliberative change while looking at a menu is not.

Claim 2: Subjects require less work to create a procedure when using only demonstrations than when using only an editor.

Claim 3: Using demonstrations and experiments results in fewer errors than when using only demonstrations.

Claim 4: Using only demonstrations results in fewer errors than when using only an editor.

Demonstrations should be helpful because Diligent uses them to identify preconditions. When identifying preconditions, Diligent uses an heuristic bias that favors likely but potentially unnecessary preconditions. Thus, subjects who use demonstrations can focus on a small set of likely preconditions, while subjects who use an editor have to consider a large set of potential preconditions.

Claim 5: Subjects require less work to create a correct procedure when using demonstrations and experiments than when using only demonstrations.

Claim 6: Subjects require less work to create a correct procedure when using only demonstrations than when using only an editor.

Claim 7: Subjects can author in less time using demonstrations and experiments than when using only demonstrations.

Claim 8: Subjects can author in less time using only demonstrations than when using only an editor.

Because it did not seem feasible, we did not test of the benefits of hierarchical procedures or the reuse of existing procedures.

7.2 The Three Versions of Diligent

In order to test the experimental hypotheses, three versions of Diligent were created. The versions support different methods for adding steps and for specifying preconditions and

[1] Section 7.4.3 describes the dependent variables.

state changes. All versions allow subjects to edit an existing plan. The three versions are described below.[2]

- *Demonstrations and Experiments.* Subjects can demonstrate procedures, and Diligent can experiment on the procedures.
- *Demonstrations.* Subjects can demonstrate procedures, but Diligent cannot perform experiments.
- *Editor Only.* Subjects cannot demonstrate and Diligent cannot experiment, but subjects can use an editor to create a declarative specification. A subject adds a step by selecting an action to perform. The subject then specifies preconditions and state changes associated with the step by selecting attributes and typing in their values. The menus for specifying actions and attribute values are only available in this version of the system.

  Requiring subjects to enter attribute values by typing is reasonable because Diligent does not know which attribute values are legal. Furthermore, typing isn't that onerous because most attribute values are short (e.g. "shut") and because subjects are given a list containing each attribute's legal values (see Appendix B). Furthermore, avoiding typing errors is a benefit of demonstrating.

  Because the subjects were given a list of all legal attribute values, one could argue that it would be little effort to provide menus containing all legal values. However, the list of legal attribute values was only provided because it was necessary for subjects who used this version of Diligent.

  However, this discussion about whether or not subjects should type in values appears to be moot because subjects appeared to make so few errors in typing that these errors had little or no effect.

  Because this version does not allow demonstrations, this version does not allow interaction with the environment while steps are being added. While steps are being added, this version ignores actions performed in the environment, does not perform actions, and ignores the state of the environment.

  This version is meant to correspond to declaratively specifying a procedure using a text editor, but unlike a text editor, this version automatically collects evaluation data, guarantees syntactic correctness and allows the system to check for consistency.

[2]Appendix D describes how to use the different versions.

For example, the system checks for consistency when deriving a procedure's step relationships. A subject is warned about inconsistency when the state changes of an earlier step establish an attribute value that is different than the value in a later step's precondition.

The menus used for all three versions are very similar. All versions use the same procedure and operator representation. The versions also use the same algorithms to derive goal conditions and step relationships. However, because the editor only version lacks knowledge of the environment's state, that version uses the preconditions and state changes of steps to create a pre-state and post-state for each step.

7.3 Usability Analysis

Prior to conducting the study, an informal analysis of Diligent's usability was conducted to ensure that the user interface and the training documentation were adequate. For the user interface, this meant that subjects could author procedures with Diligent and knew how to find various types of information. For training documentation, this meant that subjects could cover the material in 30 to 40 minutes.

In order to avoid using all potential subjects, usability was tested on only three subjects (1 graduate student and 2 research staff) over a period of a couple months.

We planned on performing the test in the following manner. Subjects would be videotaped using Diligent and would vocalize their thoughts. However, unlike a formal protocol analysis [Chi97, ES84, JH95, GMAB93], the subject's vocalizations would not be systematically analyzed. Subjects were to use the same training material as the formal evaluation. Additionally, the subjects were to learn all three versions of Diligent.

However, the test did not go as planned. Because of problems in the documentation and, to a lesser extent, the user interface, none of the subjects completed all the training material. Of the training material for the study's two sessions, the subjects only covered the first session's material.[3] Furthermore, subjects had difficulty vocalizing their thoughts. The most important finding was that first day's training took at least 50 minutes. The long training period is important because it limited the number of people willing to be test subjects.

[3] As will be explained in greater detail (Section 7.4.4), the study took two sessions, one in which the subjects learned how to use Diligent, and the second in which they reviewed what they had previously learned and then carried out the test.

Although only a few subjects were used, each subject caused substantial improvements in the materials given the next subject. This phase of the evaluation resulted in the following changes:

- Too many of the windows looked alike. One subject did not notice the window titles on the window borders. This created confusion about which menu was being viewed and about the functionality of different menus. Giving the menus large titles seemed to solve this problem.

- The interfaces of too many programs were used. Subjects interacted with four programs that each had a different look and feel. The most problems were caused by the differences between Diligent and the STEVE tutor. At the time, STEVE's interface was used for testing procedures, but the interface inconsistencies between STEVE and Diligent made the training more difficult.

  Therefore, Diligent was given control of testing. Although Diligent uses STEVE's functionality, subjects initiated testing inside Diligent. This has a number of advantages. A lot of debugging activities are combined onto one menu. The approach also supports easier instrumentation and allows Diligent to disable testing during activities such as experiments and demonstrations. Earlier, the possibility that subjects might simultaneously test and demonstrate was a major concern.

  This issue has architectural implications for this type of heterogeneous system. Either the disparate software components must conform to a common user interface look and feel, or the components need to support the use of their functionality by other components. Given that components may come from very different sources, exporting functionality may be easier than enforcing a common look and feel.

- Use as many forcing functions as possible. A *forcing function* [Nor88] prevents a user from performing actions that are unwanted in a given context. For example, Diligent's windows contain buttons that will close them, but one subject kept using the X-window exit command. This behavior was unanticipated and caused inconsistent data. This problem was solved by preventing the X-window exit command from closing the window. Another example of a forcing function is disabling testing during a demonstration.

- The user's manual was transformed into a tutorial. Originally, the user's manual gave instructions for a running example while extensively describing each window. Unfortunately, subjects had difficulty remembering the important points. Therefore, the user's manual was transformed into a tutorial by trimming unnecessary descriptions, adding summaries and ignoring unnecessary windows. Surprisingly, most of the effort to improve usability involved improving the tutorial.[4]

7.4 Experimental Method

After analyzing and improving Diligent's usability, a study was conducted to evaluate Diligent. The study had a between-subjects design where each subject authored two procedures and used only one version of Diligent.

7.4.1 Independent Variable

The independent variable was the method of authoring. Each of the three versions of Diligent (Section 7.2) represented a different method of authoring. Thus, there were three experimental conditions.

- EC_1: Authoring with demonstrations and experiments.
- EC_2: Authoring with only demonstrations.
- EC_3: Authoring with only an editor.

As mentioned earlier, all three experimental conditions allowed subjects to edit existing procedures.

7.4.2 Test Subjects

Test subjects were recruited by asking computer science graduate students and sending email to the staff at the Information Sciences Institute. Sixteen subjects started the study, and all but one finished it.[5] Of the fifteen subjects who completed the study, fourteen were computer science graduate students and one was a member of the technical staff.[6] Most subjects work in areas related to artificial intelligence.

Subjects were paid 20 dollars.

An effort was made to balance the subjects' sex, education and whether they were native English speakers. However, this proved difficult because few subjects were available

[4]While the tutorial covered authoring an example procedure in a keystroke by keystroke manner, Diligent was not used to directly author the tutorial because Diligent cannot capture screen snapshots of its own menus.

[5]The subject who quit felt that he was too busy to finish the study.

[6]It was initially thought that the subject was a graduate student.

and willing, because subjects cancelled, and because of problems that resulted in some lost data for the first procedure. (We will discuss these problems later.)

The fifteen subjects were placed in the three groups in an uneven manner. Groups EC_1, EC_2 and EC_3 had 4, 6 and 5 subjects, respectively. One factor influencing this was the inability to collect some data for EC_2 subjects. Another factor was that few subjects were available later in the evaluation. One subject (subject 11) was switched from EC_1 to EC_2 because the subject used demonstrations but no experiments.[7]

Type of subjects	**Group**		
	EC_1	EC_2	EC_3
male native speaker	0	1	3
male non-native speaker	3	3	2
female native speaker	1	1	0
female non-native speaker	0	1	0

Table 7.1: Distribution of Subjects Based on Sex and Language

Table 7.1 shows the distribution of subjects based on sex and language. The major balancing effort was attempting to get enough subjects in each group. The next criteria was trying to balance English ability and then sex. Furthermore, if it was felt that a subject knew that Diligent uses programming by demonstration, then the subject was put in groups EC_1 or EC_2. This was done to avoid preconceptions from biasing the control group (EC_3).[8]

Because subjects had to cover around 90 pages of training material, it was felt that native English speakers would find the training easier. For this reason, subjects were distributed so that no group had more English speakers than the control group (EC_3).

One problem with the methodology is that the background questionnaire was filled out after the subjects were assigned to an experimental condition. This means that only sex and English ability were immediately obvious. Thus, the number of years of education could only be roughly estimated and was, therefore, difficult to use for assigning subjects to groups.

[7]In order to keep Diligent's user interface responsive, Diligent only experiments when asked to do so by the user.

[8]For the last few subjects, whether a subject was likely to know that Diligent uses programming by demonstration was not considered.

7.4.3 Dependent Variables

The goal of the experiment was get some measure of the difficulty in authoring. The idea is that authoring should be faster and more accurate if there is less burden placed on the instructor.

The dependent variables were

- *Time.* Time was measured three ways. One time was the training time, which includes the few minutes used to fill out the background questionnaire. The second time was the time spent authoring before the subject started testing the procedure, and the third time was the total time spent authoring a procedure.

  After a subject started testing, the subject still could provide demonstrations and edit procedures, and Diligent could still perform experiments.

- *Logical Edits.* A *logical edit* is an authoring activity that requires knowledge of the procedure or the domain. Logical edits can be thought of as deliberative changes to Diligent's knowledge base. Logical edits were used to factor out time-related user interface efficiency issues that were highly dependent on the structure and layout of menus. The follow items were counted as logical edits:[9]

  – Adding or demonstrating a step.
  – Performing an action as part of a demonstration's prefix.
  – Deleting a step from a procedure.
  – Editing preconditions, state changes, goal conditions, and step relationships (i.e. causal links and ordering constraints).
  – Edits to a filter. A filter allow a subject to prevent a given attribute from being used in causal links or ordering constraints.[10]

  Logical edits did not include more passive activities, such as looking at menus or approving data derived by Diligent. (Diligent uses its knowledge of preconditions and state changes to derive a procedure's goal conditions and step relationships.)

  Logical edits were recorded in two places: immediately before a subject started testing a procedure and when a subject was finished with a procedure. During

[9]Edits to associate an effect with a step were also measured for EC_3 but were not used because these edits usually required little thought.

[10]Filters are meant to remove "nuisance" attributes that an author doesn't care about. Filters were not needed in the procedures being authored, and none of the subject's used them.

authoring, Diligent automatically collected the metrics used for counting logical edits. After a procedure was finished, the metrics and Diligent's knowledge base were saved to files. After saving the data, Diligent prepared for the next procedure by erasing its knowledge base and clearing the counters used for gathering metrics.

- *Errors.* Another metric was number of errors in a procedure's plan. Each plan was compared against an ideal target plan. Each additional or missing piece of knowledge was counted as one error. See Section 7.4.3.1 for details on how errors were measured.

- *Total required effort.* This was the amount of work needed to make a procedure correct. This was the sum of logical edits and errors. For simplicity, we assumed that each error could be corrected by one logical edit.

- *Qualitative Impressions.* After authoring both procedures, subjects filled out a questionnaire about their subjective impressions of Diligent.

7.4.3.1 Measuring Errors in Plans

When a subject finished authoring a procedure, the procedure's plan was saved to a log file. After all subjects had completed the study, the subjects' plans were compared against idealized target plans (Appendix B). This comparison identified errors in the subjects' plans.

Errors occur when a plan has missing or unnecessary steps or step relationships. The problem is that it is sometimes difficult to count errors. For example, a plan may have a necessary step that is repeated several times. Obviously, the step should only be counted once. However, the subject's plan may contain all the causal links for the target plan's step without containing a single step that is associated with all the causal links. The issue is how to decide which step relationships are correct.

Errors were calculated as follows.

- Each difference from the target plan counted as one error. In other words, each incorrect or missing step, causal link or ordering constraint counted as one error.

- A step was correct if the target plan contained a step with the same action. However, each step in the target could only match a single step in the subject's plan. When multiple steps in subject's plan mapped to a step in the target, one of the subject's steps was selected based on the comparison of its relative position compared to the plan's other steps. In particular, the step was chosen to preserve as many dependencies between steps as possible.

In several instances with the editor only version (EC_3), the state changes of several target steps were produced by a single step. When this happened, the subject's step was associated with the target step that seemed most reasonable.

- Causal links and ordering constraints were checked by comparing corresponding steps in the target and subject's plans.

  Causal links and ordering constraints could also be matched if only one of their two steps mapped to a step in the target procedure. In this case, the action of the excluded step must have matched an action of one of the steps in the target procedure. However, a step relationship (i.e. causal link or ordering constraint) in the target plan could only match one step relationship in the subject's plan.

  Counting a step relationship when only one matching step helped when a subject's procedure contained an unnecessary repetition of one of the target procedure's steps. This most often helped plans by subjects that only used an editor (EC_3).

- When authoring a procedure, subjects sometimes authored several plans. When this happened, the plans were inspected and the most complete and correct plan was used. (This always appeared to be the most recent plan.) However, the logical edits for all the plans were counted.

- The final version of plans were also inspected to see if STEVE could demonstrate them. Things that prevented successful demonstrations include

  - Missing steps.
  - Some of the step relationships used in the plan would not be satisfied by the environment.

  A demonstration was considered possible if the order of the steps in the procedure was valid, even if some of the necessary step relationships were missing.

7.4.4 Test Procedure

To perform the study, the subjects completed the activities listed in Table 7.2. The activities took place over two consecutive days. Each day's activities took approximately two hours.

Participation of each subject covered two days so that subjects could assimilate the first day's training. All subjects appeared more proficient on the second day.

Day	**Activity**	**Time Limit (minutes)**
1	Fill out background questionnaire.	
	Work through the first day tutorial. Read short system overview. Manipulate the environment's graphical interface. Read about procedural representation and fill out procedural representation worksheet. Create a procedure. Edit and test the procedure. Review summary of how to use Diligent.	
2	Review the first day tutorial by focusing on the summary	10
	Work through the second day tutorial. Review the first day's material by authoring a simple procedure. Learn how to delete unwanted steps.	
	Solve a practice problem, which involves creating and testing a procedure.	10
	Look at practice problem solution	
	Start experiment by authoring the first procedure.	30
	Author the second procedure	30
	Fill out questionnaire about impressions of Diligent	

Table 7.2: Activities Performed By Subjects

The training received by all experimental conditions was deliberately very similar. The training for the group with both demonstrations and experiments (EC_1) was nearly identical to the group that allowed demonstrations but no experiments (EC_2). Even the training for the group who could only use an editor (EC_3) was similar to the other groups. In fact, the tutorial for EC_3 differed from the other groups only in how steps were added to a procedure and how preconditions and state changes were specified.

The tutorial starts out very specific but becomes more abstract after an activity has been described. When first encountering an activity, the tutorial describes each action on a button click by button click basis. Associated with these detailed instructions were dozens of screen snapshots. Later, after an activity has already been covered, the tutorial only provides a high level description of what needs to be done. The initial detail provides scaffolding that promotes initial understanding, and the later removal of the scaffolding promotes competence by reducing the reliance on detailed instructions. For example, the first day tutorial for EC_1 is 77 pages[11] and has over 48 figures and tables. In contrast, the second day tutorial reviews much of the same material in only 7 pages.

Before authoring a procedure on the 1st day (Table 7.2), subjects read about the procedural representation and filled out a worksheet on it (Sections B.2 and B.3). This separation was critical for training. Otherwise, users would be required to author a procedure before they understood the procedural representation. If subjects were to focus on learning the representation, they might pay less attention to learning the user interface. The material on procedure representation was believed to be much more important for group that used an editor instead of demonstrations (EC_3).

The practice problem at the end of training helped ensure that subjects were ready to perform the experiment. The problem let subjects use the system without directions, and the solution allowed them to check if they had misconceptions about how they should author.

Originally, the test monitor was to have little or no communication with a subject that was not part of the test script. Questions would be answered by pointing to windows or pre-printed answers, such as "yes" or "no." However, this proved very awkward. Therefore, during training, pointing to windows and tutorial pages was used, but verbal answers (e.g. "yes") were sometimes given. An effort was made to make verbal answers as short and as specific as possible. Questions were only answered if they were relevant to the tutorial material that the subject was working on. Because of the detail in the tutorial, questions were infrequent. In contrast, during the experiment, pre-printed directions were used and questions could not be answered. However, if there was a software problem during an experiment, the test monitor spoke to the subject and attempted to put the system back into a usable state.

Before authoring each procedure, Diligent's existing knowledge of the domain was erased. The environment was then placed in the procedure's initial state, and the subjects were then given the following information:

- A functional description of the procedure without an explicit specification of which steps to perform. The steps were not included because there was concern that subjects would simply transcribe the description into Diligent's representation.

- A set pictures with labels for relevant objects.

[11]Many pages contain a great deal of whitespace.

- A list of all HPAC attributes and their legal values. (The list was needed by the control group (EC_3), which could not use demonstrations.)

Although both the training and the experiment used the HPAC domain, the HPAC objects used in the experiment were not used for authoring procedures during training.

Additional information on the test procedure is contained the appendices. Appendix D contains some of the tutorial material.[12] Appendix B contains the other evaluation materials (e.g. directions). Appendix C contains deviations from the test procedure as well the other data collected during the study.

7.4.5 The Procedures Being Authored

During the experiment, subjects authored two procedures.[13] The procedures are derived from real procedures in the HPAC domain, but have been adapted to the portion of the HPAC that is supported by the graphical interface. The simulation that controlled the environment was modified so that the procedures were supported and were partially ordered.[14] The procedures were chosen for the following reasons:

- They were partially ordered.
- Knowledge of one procedure should provide little or no help on the other procedure.
- Each procedure was logically one procedure rather than a concatenation of two procedures.
- They had between 6 and 8 steps.

The two procedures have slightly different properties. The first procedure has a deliberately more abstract description and is more complex than the second procedure. (8 steps, 13 ordering constraints and 30 causal links versus 6 steps, 7 ordering constraints and 16 causal links.) The procedures were authored in this order because reversing the order might have caused subjects to include "unwanted" attributes in the more complex procedure, which would have made scoring the procedure more difficult.

[12] The training material for each of the groups contains approximately 90 pages. Because the length and the similarity of the material between groups, Appendix D combines training material for the three groups.

[13] The procedures that were authored and the test materials are in Appendix B.

[14] Diligent is designed for partially ordered procedures. The experiments performed by Diligent may not learn much if a procedure is totally ordered. A procedure is *totally ordered* if there is only a single valid order for performing the steps.

There were a number of problems when subjects authored the first procedure. First, there was a memory leak that would cause the environment's graphical interface (i.e. Vista Viewer) to become less responsive and sometimes crash. This problem was fixed with a software upgrade. Second, subjects would sometimes demonstrate steps too quickly. This caused the steps to appear to be simultaneous, and simultaneous steps cause problems with Diligent's operator learning algorithms. This problem was made more likely as the Vista Viewer became less responsive. The problem was addressed by fixing the memory leak and by reminding subjects not to demonstrate too quickly. An additional problem was that the description of the first procedure was unclear. This caused some subjects to have difficulties identifying the correct steps. This problem was addressed by clarifying the description.

Because of these problems, different numbers of subjects are used when analyzing the two procedures. Only the final 6 subjects are used for the first procedure, while all subjects are used for the second procedure.

7.4.6 Data Analysis

Section 7.1 contains testable claims about differences between groups EC_1 and EC_2 and between groups EC_2 and EC_3. To test for differences between groups, we used Analysis of Variance (ANOVA) [WW72], which tests for the differences between all groups. ANOVA compares variance within groups to variance between groups.[15]

Because ANOVA depends on groups having a normal distribution and similar variances, we also used the Kruskal-Wallis test. The Kruskal-Wallis test is a non-parametric test, which means that it does not depend on the distribution. Instead of using a dependent variable's values, the Kruskal-Wallis test sorts the values and uses their relative order. Of course, a non-parametric test requires greater differences than an ANOVA test.

Because ANOVA and Kruskal-Wallis compare all groups, we performed post hoc tests to compare pairs of groups. A post hoc test simultaneously compares pairs of groups to identify significant differences while maintaining a 95% probability that all comparisons are true. The post hoc test used was Scheffé's F [Sch53], which requires a significant ANOVA F value but is robust with in regard to heterogeneous variances.

For statistical significance, we used the .05 probability level.

[15]All statistical calculations were performed with version 5.0 of StatView for Windows by SAS Institute Inc..

A word of caution – the statistical significance of this chapter's results should be viewed with a little skepticism. Significance was difficult to establish because there were few subjects. Furthermore, because there were so few data points, the results are too sensitive to individual data points. In fact, some researchers do not consider data as statistically significant unless there several times as many subjects as in this study. Nevertheless, the results are valuable because they suggest patterns and trends.

7.5 Results

This section presents the data collected during the study.[16] (The data will be discussed in Section 7.6.)

In the following tables, a few conventions are used. The number of digits shown may not indicate the number of significant digits. The "pre-test" values are the values when subjects started testing their procedures. If a subject didn't test a procedure, the final value was used.

As mentioned earlier, because there were relatively few subjects, the following data are used to suggest trends and patterns rather than to provide solid statistical proof.

7.5.1 Results of Background Questionnaire

At the beginning of training, subjects filled out a background questionnaire. Their responses were then analyzed to look for patterns in the distribution of subjects between groups. The experimental condition (e.g. EC_1) was used as an ANOVA factor for this analysis. The results are shown in tables 7.3 and 7.4.[17]

The only significant difference is the typical time spent browsing per week. The group that demonstrated without experiments (EC_2) spent the most time browsing (13 hours).

The variable *education* represents years of education. This includes 12 years for graduating from high school. The group that demonstrated and experimented (EC_1) was the oldest and the group that only demonstrated (EC_2) was the youngest. However, the standard deviation of the group that only used the editor (EC_3) is several times larger than the standard deviations of the other groups.

The variable *English ability* indicates a subject's rating of his English proficiency. The subject's rating was converted into a numeric value: good (1), excellent (2), native (3).

[16] Appendix C contains a more detailed presentation of the data collected during the study.

[17] The ANOVA values were computed with 12 and 2 degrees of freedom, except for previous week's computer use, which had 11 and 2 degrees of freedom.

Dependent Variable	**F**	**Probability**
Education	1.542	.2535
English ability	1.346	.2969
Sex	0.912	.4278
Age	0.863	.4466
Machine learning knowledge	0.340	.7187
Artificial intelligence planning knowledge	0.340	.7187
Programming ability	0.167	.8484
Typical browsing	4.851	.0286
Programming last week	3.806	.0554
Typical hours/week	2.211	.1522
Browsing last week	1.928	.1916
Total hours last week	1.774	.2149

Table 7.3: Background ANOVA Tests

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Education	21.9	0.85	18.9	1.5	19.5	4.3
English ability	1.7	0.96	2.0	0.89	2.6	0.55
Sex	0.75	0.5	0.67	0.5	1.0	0
Age	33.7	2.5	30.0	3.2	35.0	10.6
Machine learning knowledge	0.5	0.58	0.3	0.52	0.6	0.55
Artificial intelligence planning knowledge	0.5	0.58	0.3	0.52	0.6	0.55
Programming ability	2.0	0.0	2.0	0.63	2.2	0.84
Typical browsing	8	6	13	5	5	0
Programming last week	8	9	20	10	31	15
Typical hours/week	40	0	49	11	44	9
Browsing last week	10	7	7	6	3	2
Total hours last week	30	8	37	14	45	7

Table 7.4: Background Means and Standard Deviations

The variable *sex* indicates whether a subject is male (1) or female (0). Because several female subjects canceled, the distribution of females is skewed.

The variable *age* is the subject's age in years. Because the questionnaire asked subjects to circle a range of ages, the top age in the interval was used.[18] The reason that group EC_3 had the largest age is that the group had the oldest subject (50).

The variables *machine learning knowledge* and *artificial intelligence planning knowledge* represent a yes (1) or no (0) about whether a subject felt he had significant knowledge in that area.

The variable *programming ability* contains a subject's self rating. A subject's rating was converted into a numeric value: intermediate (1), good (2), expert (3).

The "typical" computer use numbers represent typical hours per week spent using a computer, and the "last week" numbers reflect the hours spent during the previous week on a computer.

7.5.2 Time Spent Training

We looked for correlations between the subjects' backgrounds and training time. The data are shown in tables 7.5 and 7.6.

We expected all groups to have similar training times because all groups received very similar training. As expected, no significant difference between groups was found for training time.

The first day's training time had more variation than the second day's training. The decrease in variation on the second day was expected because less material was covered and because the subjects were already familiar with the system.

Dependent Variable	**F**	**Probability**
Day 1	0.916	.4265
Day 2	0.281	.7598
Total time	0.762	.4881

Table 7.5: Background ANOVA Tests

In order to find correlations between training time and background variables, a multiple linear regression was performed. During the regression, a subject's experimental condition was ignored, and the total training time was used as the dependent variable. The best fit

[18]A range was used because one usability test subject complained about asking for an exact age.

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Day 1 (min.)	115	25	97	19	97	26
Day 2 (min.)	44	8	43	14	39	10
Total Time (min.)	159	23	140	28	136	35

Table 7.6: Training Time Means and Standard Deviations

that was found is shown in Table 7.7. Three independent variables were identified: years of education, artificial intelligence (AI) planning knowledge and English proficiency. Of the independent variables, only English proficiency was expected, and unlike the other independent variables, English proficiency is not statistically significant (P-Value). The regression coefficients (Std. Coeff.) indicate that English proficiency and AI planning knowledge decrease training time, while more education increases training time. The R^2 (R Squared) indicates that the independent variables only predict 61 percent of the variation in training time.

7.5.3 Logical Edits

While subjects authored procedures, Diligent recorded the number of logical edits that they performed. A logical edit is an authoring activity that requires knowledge of the procedure or the domain (e.g. demonstrating a step). Logical edits do not include passive activities, such as looking at menus or approving data derived by Diligent. Instead, a edit is deliberative change to Diligent's knowledge base.

The data from the analysis are shown in Table 7.8, and graphs of the data are shown in Figure 7.1. The "pre-test" value is the value when the subjects started testing their procedures.

Procedure 1's results are weak because only 6 six subjects were used. No significant differences between groups were found, but the values for EC_1 (demonstrations and experiments) are much smaller than for the other two groups.

Procedure 2's results are stronger. There is a significant difference between the groups both before and after testing (ANOVA and Kruskal-Wallis). There is also a significant difference between groups EC_1 and EC_3 and between groups EC_2 and EC_3. The group that used demonstrations and experiments (EC_1) had the lowest values, while the group that used an editor (EC_3) had the highest values.

Regression Summary
total training vs. 3 Independents

Count	15
Num. Missing	1
R	.781
R Squared	.609
Adjusted R Squared	.503
RMS Residual	20.483

ANOVA Table
total training vs. 3 Independents

	DF	Sum of Squares	Mean Square	F-Value	P-Value
Regression	3	7195.687	2398.562	5.717	.0131
Residual	11	4615.247	419.568		
Total	14	11810.933			

Regression Coefficients
total training vs. 3 Independents

	Coefficient	Std. Error	Std. Coeff.	t-Value	P-Value
Intercept	83.291	54.617	83.291	1.525	.1555
education	4.915	2.185	.471	2.249	.0460
English ablity	-11.171	7.706	-.321	-1.450	.1751
Planning know ledge	-28.569	11.356	-.508	-2.516	.0287

Table 7.7: Linear Regression on Total Training Time

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 total edits	9.5	2.1	35.0	12.7	37.5	7.8
Procedure 2 pre-test edits	8.7	2.1	11.8	5.1	24.6	4.6
Procedure 2 total edits	9.0	2.2	16.8	6.4	26.0	3.8

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 total edits	6.346	.0836
Procedure 2 pre-test edits	18.048	(*) .0002
Procedure 2 total edits	14.021	(*) .0007

Kruskal-Wallis Results

Dependent Variable	**Probability**
Procedure 1 total edits	.1801
Procedure 2 pre-test edits	(*) .0079
Procedure 2 total edits	(*) .0070

Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 total edits	.1316	.1064	.9601
Procedure 2 pre-test edits	.5599	(*) .0006	(*) .0014
Procedure 2 total edits	.0785	(*) .0008	(*) .0273

Table 7.8: Logical Edit Analysis

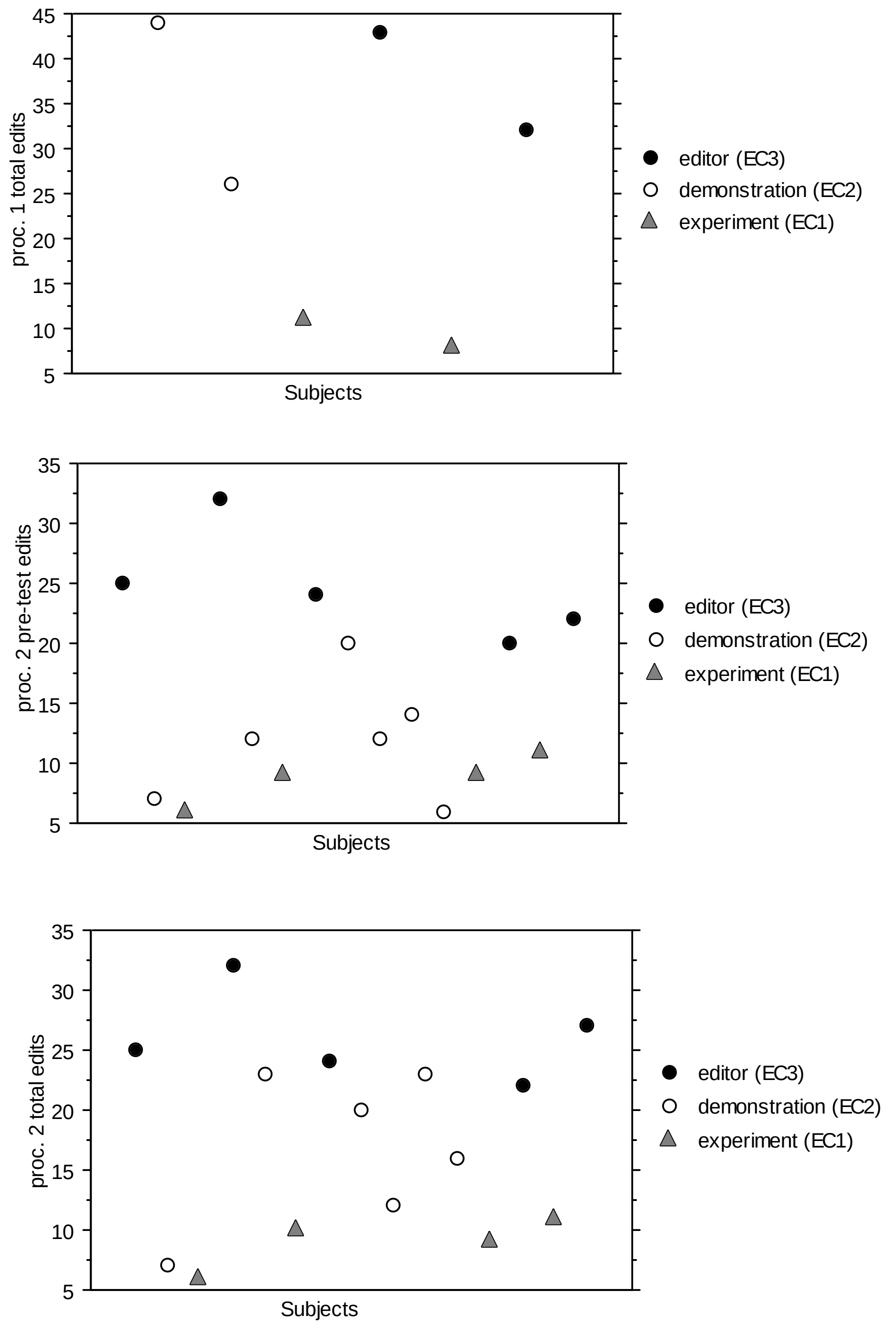


Figure 7.1: Graphs of Logical Edits

7.5.4 Errors

We will now present data on the errors in the subjects' procedures. This has several aspects: how well were the subjects able to determine a procedure's steps; the number of components (e.g. causal links) missing from a procedure; and the number of unnecessary components in a procedure. We will finish by looking at the total errors.

7.5.4.1 Errors in Identifying Steps

An important influence on the number of errors is how many of the procedure's steps are incorrect. This is important because any step relationships involving a missing or unnecessary step will be counted as errors. It was expected that subjects would have little difficulty in correctly identifying steps.

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 final missing steps	0	0	0	0	3.5	.7
Procedure 1 final unnecessary steps	0	0	1.5	.7	1	1.4
Procedure 2 final missing steps	1.0	.8	0	0	.2	.4
Procedure 2 final unnecessary steps	0	0	.2	.4	.4	.9
Procedure 1 final invalid steps	0	0	1.5	.7	4.5	.7
Procedure 2 final invalid steps	1	.8	1	2	.6	.9
Procedure 1 works	1	0	1	0	0	0
Procedure 2 works	.250	.5	.833	.4	.4	.6

Table 7.9: Means and Standard Deviations on Invalid Steps

How well the subjects were able to determine which steps to perform is shown in Table 7.9. The values in the table represent the final versions of the procedures. The "invalid steps" are the total missing and unnecessary steps. The last two rows (e.g. Procedure 1 works) indicate whether a valid sequence of steps was specified (1 means yes and 0 means no).

The biggest difference in the number of errors was for Procedure 1. A significant difference between the groups was found (ANOVA at a 1% level). The post hoc tests

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 final errors	4.5	.7	9	4	44	.7
Procedure 2 pre-test errors	11.5	8.5	3.5	4	11.6	5
Procedure 2 final errors	11.5	8.5	6	7	11.8	5

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 final errors	151.605	(*) .0010
Procedure 2 pre-test errors	3.388	.0681
Procedure 2 final errors	1.268	.3164

Kruskal-Wallis Results

Dependent Variable	**Probability**
Procedure 1 final errors	.1017
Procedure 2 pre-test errors	.1054
Procedure 2 final errors	.3371

Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 final errors	.3368	(*) .0013	(*) .0018
Procedure 2 pre-test errors	.1504	.9978	.1157
Procedure 2 final errors	.4739	.9978	.3949

Table 7.10: Errors of Omission Analysis

indicates a significant difference between EC_3 and the other groups. The groups that used demonstrations (EC_1 and EC_2) had fewer invalid steps.

The differences between groups for Procedure 2 are relatively minor, and no significant differences were found. When considering the percentage of procedures that would have worked, the subjects in EC_2 did a better job of demonstrating than the subjects in EC_1.

7.5.4.2 Errors of Omission

If a plan is missing a component (i.e. step relationship or step), the error is called an error of omission.

The data from the analysis are shown in Table 7.10, and graphs of the data are shown in Figure 7.2.

Because Diligent's heuristics favor errors of commission, one would expect the group that used an editor (EC_3) to have more errors of omission.

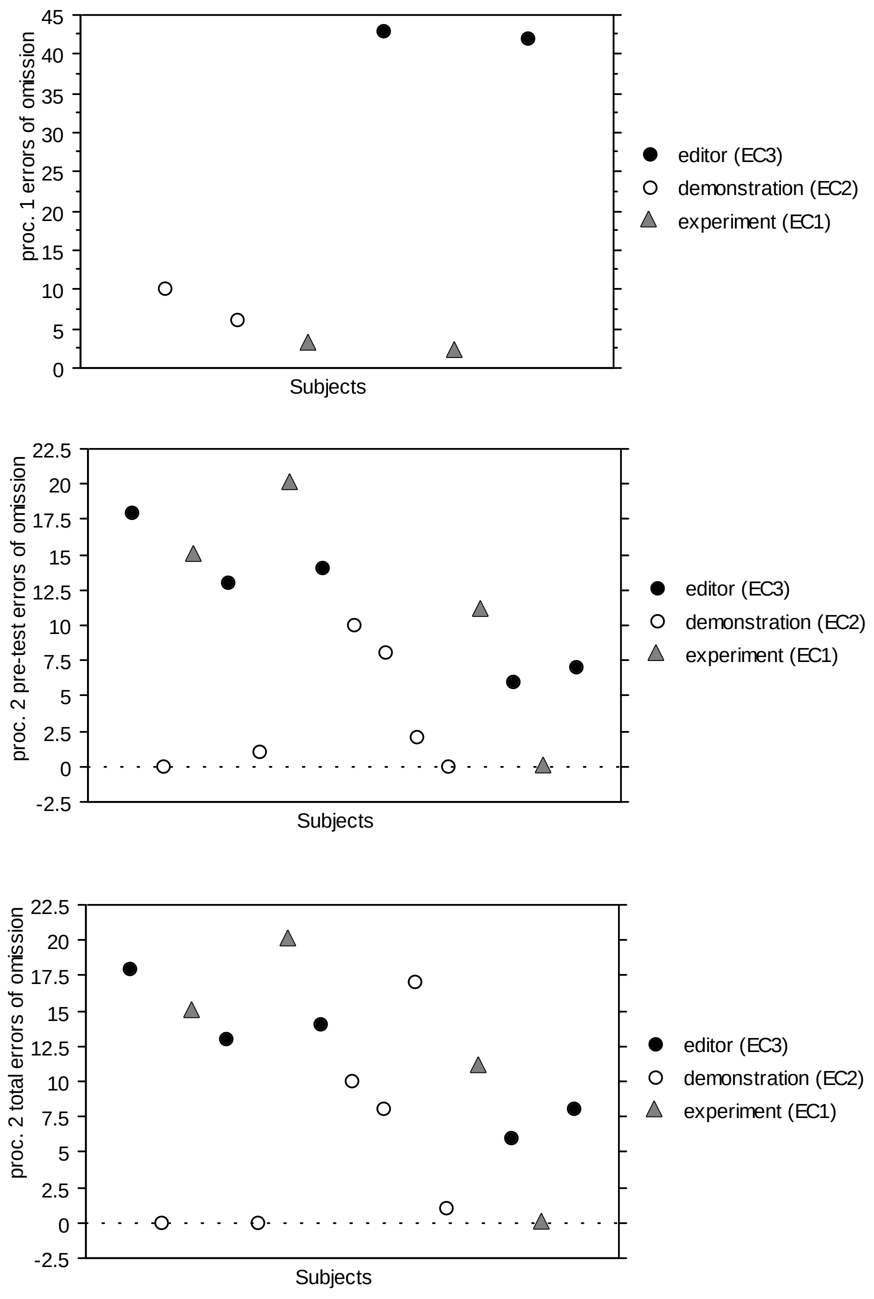


Figure 7.2: Graphs of Errors of Omission

In Procedure 1, the groups are significantly different (ANOVA). Group EC_3 has more errors and is significantly different than the other groups. Group EC_1 has slightly fewer errors than group EC_2.

In Procedure 2, there is no significant difference between the groups. However, group EC_2 did better than the other groups. This was unexpected because EC_1 and EC_2 both used demonstrations.

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 final errors	6.5	.7	26	18	8.5	.7
Procedure 2 pre-test errors	4	4	12	5	5	8
Procedure 2 final errors	4	4	10	6	6	8

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 final errors	2.037	.2762
Procedure 2 pre-test errors	2.843	.0975
Procedure 2 final errors	1.015	.3916

Kruskal-Wallis Results

Dependent Variable	**Probability**
Procedure 1 final errors	.1017
Procedure 2 pre-test errors	.0712
Procedure 2 final errors	.3566

Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 final errors	.3236	.9826	.3808
Procedure 2 pre-test errors	.1524	.9577	.2020
Procedure 2 final errors	.4122	.8765	.6750

Table 7.11: Errors of Commission Analysis

7.5.4.3 Errors of Commission

If a plan has an unnecessary component (i.e. step relationship or step), the error is called an error of commission.

The data from the analysis are shown in Table 7.11, and graphs of the data are shown in Figure 7.3.

Diligent's heuristics favor errors of commission over errors of omission because it should be easier for an instructor to identify a mistake among a small set of unnecessary items than among a large set of missing items. Thus, we would expect group EC_2 to have the most errors. Group EC_1 should have fewer errors than EC_2 because experiments should remove unnecessary conditions. Group EC_3 should also have few errors because subjects have to explicitly specify each unnecessary item.

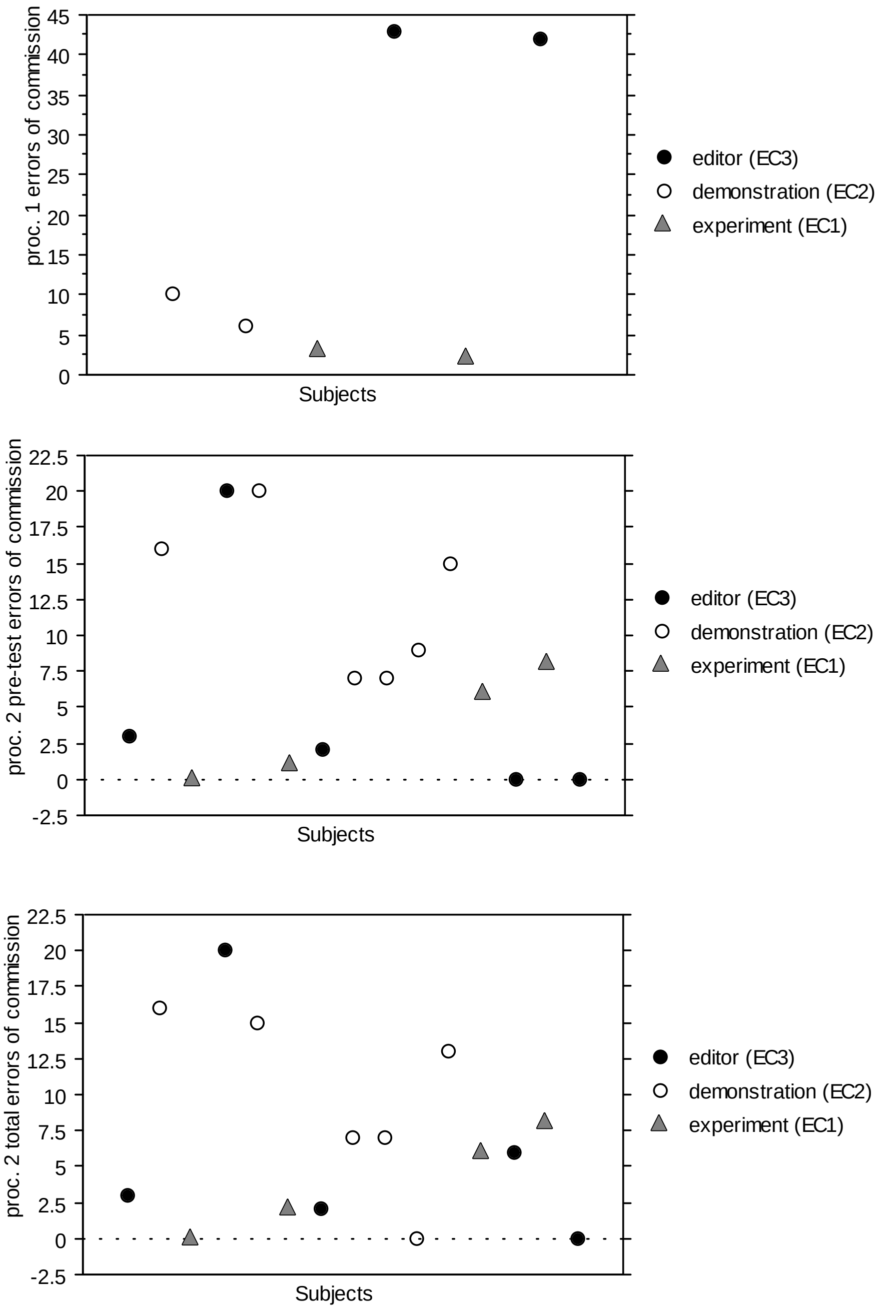


Figure 7.3: Graphs of Errors of Commission

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 final errors	11	1	35	14	53	1
Procedure 2 pre-test errors	15	5	16	3	17	11
Procedure 2 final errors	15	6	16	1	18	10

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 final errors	13.059	(*) .0331
Procedure 2 pre-test errors	.039	.9617
Procedure 2 final errors	.240	.7906

Kruskal-Wallis Results

Dependent Variable	**Probability**
Procedure 1 final errors	.1017
Procedure 2 pre-test errors	.9972
Procedure 2 final errors	.9842

Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 final errors	.1338	(*) .0334	.2402
Procedure 2 pre-test errors	.9923	.9626	.9850
Procedure 2 final errors	.9992	.8432	.8337

Table 7.12: Total Error Analysis

Procedure 1 has a large number of step relationships. Thus, one would expect a large number of errors for group EC_2, while groups EC_1 and EC_3 would have few errors. This is what was found. However, there were no significant differences between the groups.

Procedure 2 is simpler than Procedure 1. Thus, all groups should have fewer errors. This is what was found. Although group EC_2 had slightly more errors than the other groups, there were no significant differences between the groups.

7.5.4.4 Total Errors

A plan's total errors are the sum of its errors of omission and commission.

The data from the analysis are shown in Table 7.12, and graphs of the data are shown in Figure 7.4.

Because Procedure 1 is the more complicated procedure, we expect the differences between groups to be larger. The groups are significantly different (ANOVA), and groups

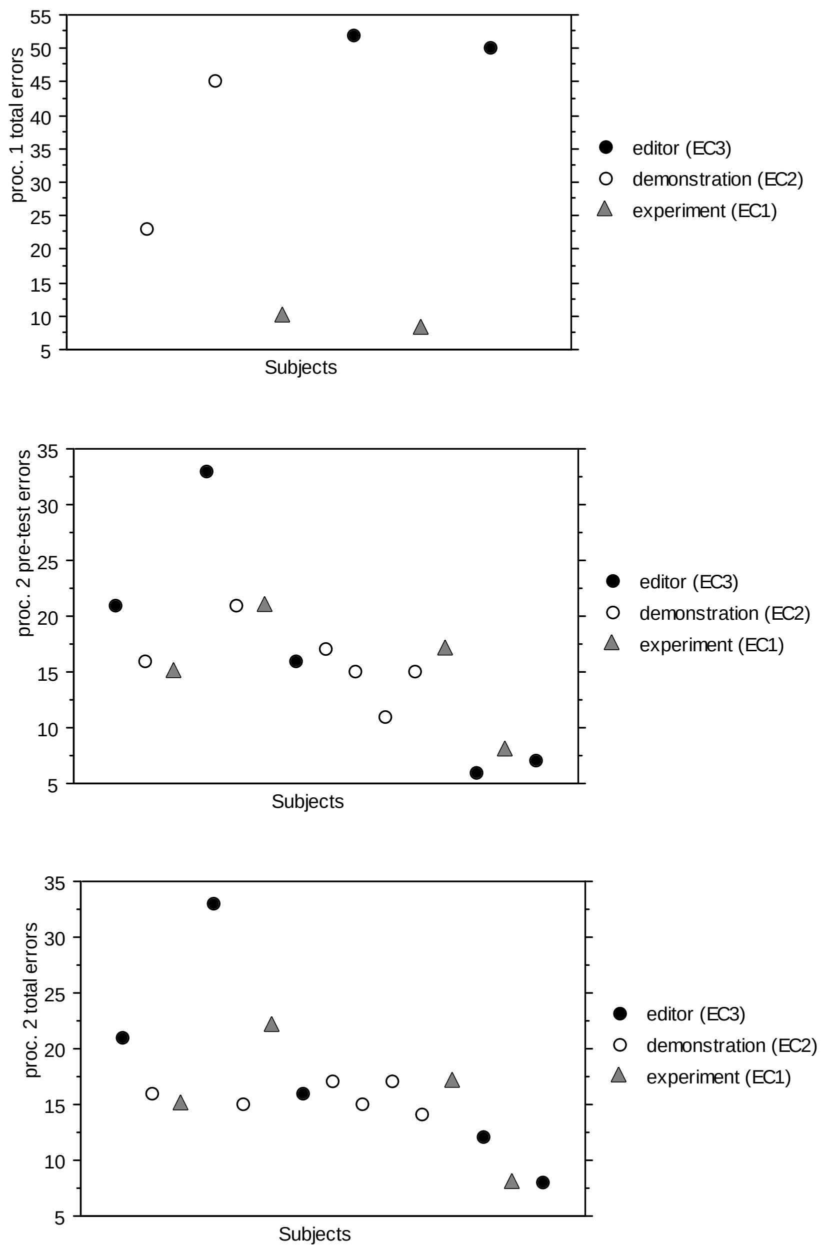


Figure 7.4: Graphs of Total Errors

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 final required effort	20	4	70	1	90	9
Procedure 2 pre-test required effort	24	5	28	6	41	16
Procedure 2 final required effort	24	6	32	7	44	13

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 final required effort	78.490	(*) .0026
Procedure 2 pre-test required effort	3.803	.0526
Procedure 2 final required effort	5.370	(*) .0216

Kruskal-Wallis Results

Dependent Variable	**Probability**
Procedure 1 final required effort	.1017
Procedure 2 pre-test required effort	.0775
Procedure 2 final required effort	(*) .0238

Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 final errors	(*) .0077	(*) .0028	.0833
Procedure 2 pre-test errors	.8565	.0771	.1301
Procedure 2 final errors	.4162	(*) .0237	.1517

Table 7.13: Total Required Effort Analysis

EC_1 and EC_3 are significantly different. The group that demonstrated and experimented (EC_1) did better than the other groups. Most errors for group EC_2 were errors of commission, while most errors for groups EC_1 and EC_3 were errors of omission.

In Procedure 2, each group had roughly the same number of total errors, and no significant differences between the groups were detected. Groups EC_1 and EC_3 had mostly errors of omission, while group EC_2 had mostly errors of commission. Group EC_1 did a little better than the other groups even though its subjects did the worst job of identifying the procedure's steps. (Poor demonstrations caused group EC_1 to have errors of omission.

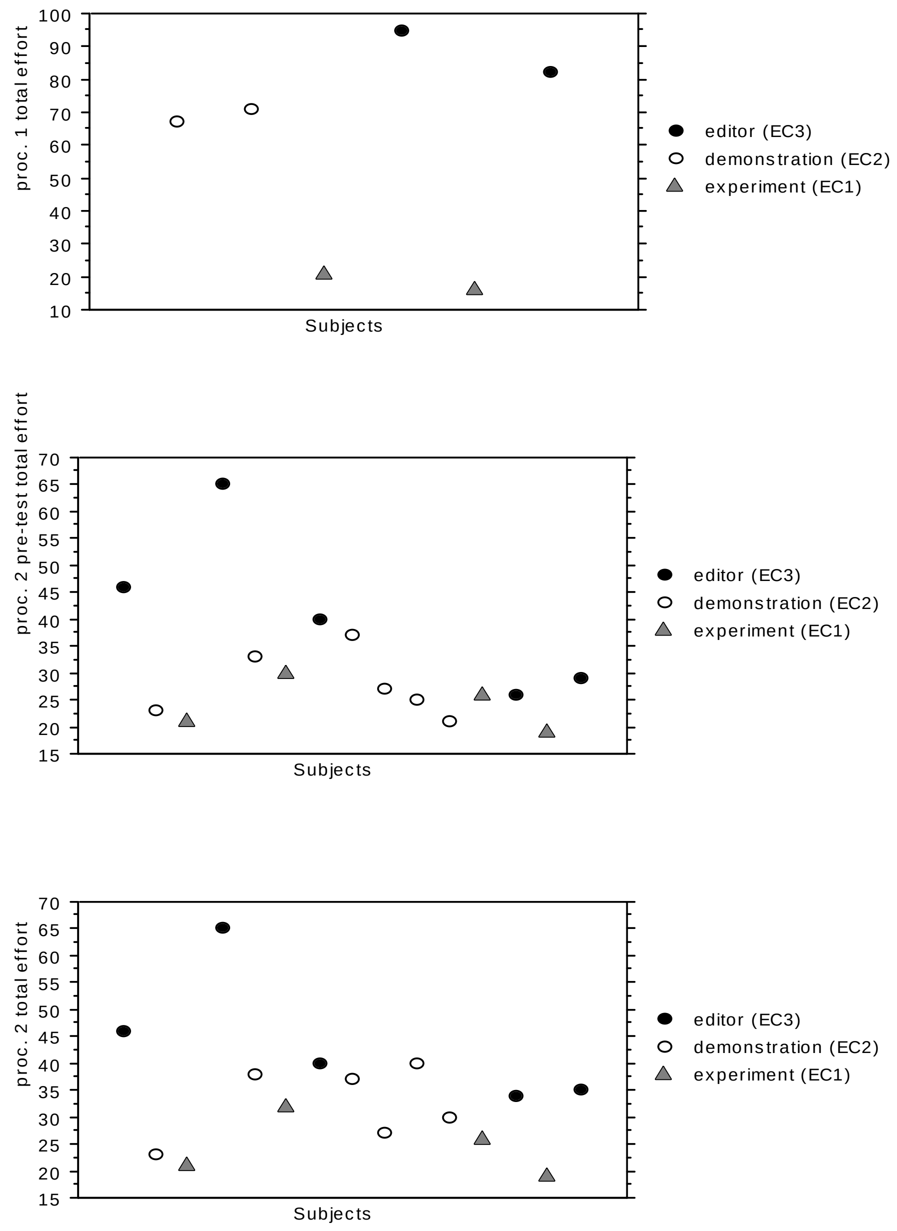


Figure 7.5: Graphs of Total Required Effort

7.5.5 Total Required Effort

The total required effort is a measure of the amount of work required to produce a correct plan. The required effort includes the work that has been done as well as the work that needs to be done. The previous work is measured by logical edits, and the additional work is estimated by the total errors. Total errors is used because each error can be corrected by an edit.

The data from the analysis are shown in Table 7.13, and graphs of the data are shown in Figure 7.5.

Procedure 1, the more complex procedure, has significant differences (ANOVA) between the groups. Group EC_1 is significantly different and better than groups EC_2 and EC_3. Group EC_2 is not as good as EC_1 but better than EC_3. The differences between the groups result from differences in both the logical edits and the total errors.

In Procedure 2, the differences between groups are close to significant before testing and are significant (ANOVA and Kruskal-Wallis) after testing. Like Procedure 1, group EC_1 is significantly different and better than group EC_3, while group EC_2 is worse than EC_1 but better than EC_3. The differences between the groups result from the fewer logical edits required by groups that use demonstrations (EC_1 and EC_2).

7.5.6 Time Spent Authoring

When subjects authored procedures, two times were measured: when testing started and when the subject finished. After testing had started, subjects could still use Diligent's full capabilities for demonstrating, experimenting and editing.

The data from the analysis are shown in Table 7.14, and graphs of the data are shown in Figure 7.6. The pre-test times for Procedure 1 are included even though none of the procedures was modified after testing started.

There was a 30 minute time limit placed on each procedure, and subjects often seemed to run out of time.

None of the groups are significantly different. However, the times for group EC_1 are slightly less than the times for EC_2, and the times for EC_2 are slightly less than the times for EC_3.

7.5.7 Subjective Impressions

After the subjects finished authoring the two procedures, they filled out a questionnaire about their impressions of Diligent. The results are shown in Table 7.15.

Means and Standard Deviations

Dependent Variable	EC_1		EC_2		EC_3	
	Mean	**Std.Dev**	**Mean**	**Std.Dev**	**Mean**	**Std.Dev**
Procedure 1 pre-test time	27	4	29	2	30	.9
Procedure 1 total time	29	2	30	.4	30	.9
Procedure 2 pre-test time	21	3	22	10	25	8
Procedure 2 total time	25	6	26	9	28	5

ANOVA Results

Dependent Variable	**F**	**Probability**
Procedure 1 pre-test time	.886	.4377
Procedure 1 total time	.777	.4816
Procedure 2 pre-test time	.217	.8077
Procedure 2 total time	.212	.8118

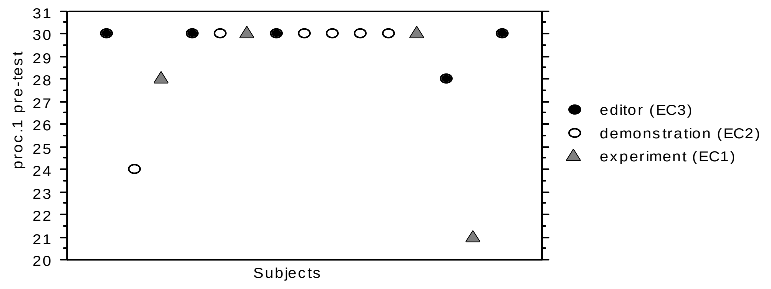
Kruskal-Wallis Results

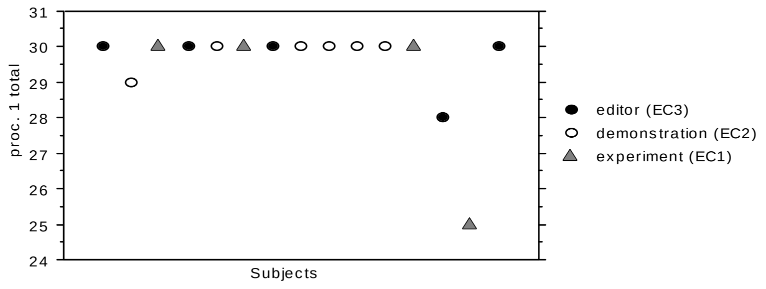
Dependent Variable	**Probability**
Procedure 1 pre-test time	.6191
Procedure 1 total time	.9370
Procedure 2 pre-test time	.7952
Procedure 2 total time	.4338

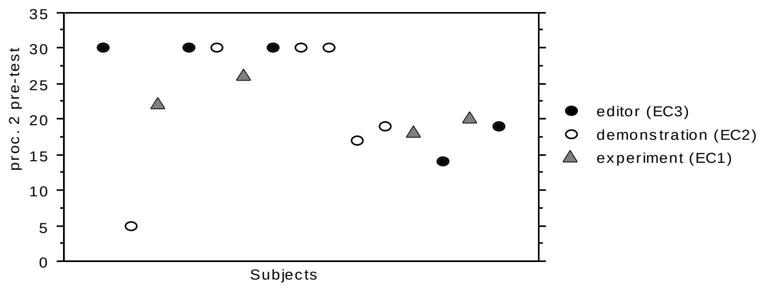
Post Hoc Test Probabilities

Dependent Variable	EC_1,EC_2	EC_1,EC_3	EC_2,EC_3
Procedure 1 pre-test time	.6178	.4567	.9356
Procedure 1 final time	.4969	.6646	.9618
Procedure 2 pre-test time	.9980	.8506	.8536
Procedure 2 final time	.9526	.8140	.9294

Table 7.14: Analysis of Time Spent Authoring







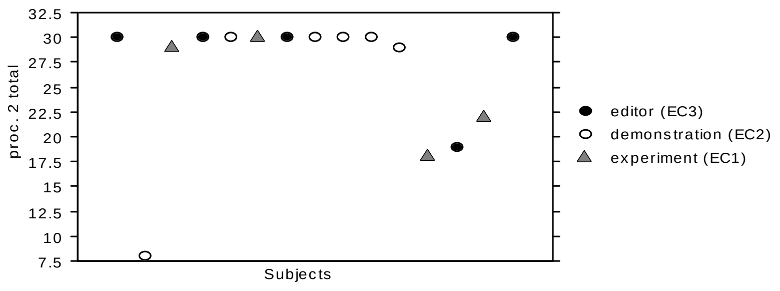


Figure 7.6: Graphs of Time Spent Authoring

Question	**Group**	**Distribution of Answers**							**Mean**
		1	**2**	**3**	**4**	**5**	**6**	**7**	
Like the system	EC_1				1	1	2		5.2
	EC_2			2	2	2			4.0
	EC_3		1	1	1	1	1		4.0
Easy to use	EC_1		1	1	1		1		3.7
	EC_2		2	1	2	1			3.3
	EC_3		2	2	1				2.8
Easy to specify a step	EC_1			1	1	1		1	4.7
	EC_2	1			2		1	2	4.8
	EC_3		1		1	2			4.4
Easy to identify preconditions	EC_1		1		1	1	1		4.2
	EC_2			2		2	1	1	4.8
	EC_3		1		1		3		4.7
Easy to identify state changes	EC_1		1		1		1	1	4.7
	EC_2			1	1	3	1		4.7
	EC_3		1		1		3		4.8
Easy to identify how operators influence preconditions and state changes	EC_1				2		2		5.0
	EC_2			4		2			3.7
	EC_3	1	1	2		1			2.8
Easy to demonstrate	EC_1		1		1			2	5.0
	EC_2	1			2		1	1	4.2
Additional demonstrations useful	EC_1	1				1		1	4.3
	EC_2			1	2	2	1		4.5
Like experimenting	EC_1	1			1	1		1	4.2
Experiments quick enough	EC_1		1				1	2	5.5
Experiments saved work	EC_1				1			2	6.0
Experiments caught errors that would have been missed	EC_1	1			2				3.0

Table 7.15: Subjective Impressions

The data under Distribution of Answers indicates how the subjects rated Diligent: 1 means not at all, 4 means somewhat, and 7 means a great deal. The numbers in a column indicate how many subjects gave that answer. Blank cells indicate that no subjects gave that answer.

7.6 Discussion

The previous section (Section 7.5) presented the study's results. In this section, we will analyze the results and discuss their meaning.

7.6.1 Assumptions About Test Subjects

Our research attempts to identify techniques that could assist domain expert instructors. By instructor, we mean someone who teaches these procedures to human students.

However, instructors were not available as test subjects. Instead, graduate students were used because they were the most available pool of subjects. In particular, we used computer science graduate students who worked mostly in fields related to artificial intelligence.

This raises the question of how similar are graduate students and instructors. To address this issue, consider the assumptions that we have made about the people who author with Diligent.

- An author is a domain expert.

  A graduate student is not domain expert, but he has access to a functional description of the procedure.

- An author knows a valid order for performing a procedure's steps.

  A graduate student has to identify a valid order of steps when given a functional description, which does not explicitly specify the order of steps. In this sense, a graduate student has a more difficult task than an instructor. In fact, some test subjects ordered steps incorrectly.

  The problem with an invalid step order is that it interferes with the heuristics that Diligent uses to create initial operator preconditions (i.e. the h-rep). Because the heuristics assume that the state changes of earlier steps are likely preconditions for later steps, mistakes in the order that steps are demonstrated interferes with the identification of likely preconditions. This suggests that the groups who used these heuristics (EC_1 and EC_2) were more likely to be negatively affected by disordered steps.

- An author may not be familiar with the simulation that models the domain. This means that he may have problems mapping his knowledge to simulation attributes. Additionally, the simulation may have some idiosyncrasies that are not obvious to the author.

  A graduate student doesn't know the domain, but his functional description should describe the necessary attributes. Like an instructor, a graduate student needs to map domain attributes to simulation attributes.

- The author is an instructor who can articulate dependencies between steps. However, he may forget to mention some dependencies.

  Not applicable to graduate students.

- The author may not be a programmer, and he may have difficulty understanding the simulation's code. He may also have problems using the rigid syntax required for declaratively specifying a procedure.

  This does not apply to graduate students. Although they have no access to the simulation's code, they all program and many of them have been exposed to declarative plan representations. Because of their prior exposure to plan representations, graduate students should learn how to use Diligent more quickly than instructors. Familiarity with the representation may also allow graduate students to use the editor only version (EC_3) more easily than instructors.

Overall, when comparing graduate students to instructors, the students should have a harder time authoring, but should learn how to use the system more quickly. Furthermore, graduate students should have an easier time using the editor only version than would instructors.

The difficulty that subjects had correctly identifying a procedure's steps (Section 7.5.4.1) lends support to the idea that the authoring task would have easier for instructors.

7.6.2 Discussion of Background Questionnaire

The first activity that subjects performed during the study was to fill out a questionnaire about their background.

Several questions asked subjects to rate themselves in area. A subject's answers seemed to depend heavily on the subject's modesty. For example, a non-native speaker's English ability did not appear to correspond to the subject's "real" English ability. Additionally, a subject's rating of his programming ability did not seem reliable, but this cannot be determined.

The number hours on a computer both last week and typically also appeared questionable because a common value was 40, which is the number of hours in a standard work week. This value seems unlikely for graduate students, who tend to keep irregular hours.

A pair of yes or no questions asked if a subject had knowledge of machine learning or artificial intelligence planning. Some subjects did not provide the desired answers: subjects who should have answered yes sometimes answered no. In hindsight, a range of values would have been better than a simple yes or no.

The only significant variable was typical hours spent browsing. There is no obvious reason why computer use numbers should influence the results of the experiment, especially since all subjects are experienced computer users.

7.6.3 Discussion of Training Time

Multiple linear regression identified three variables that seemed to influence training time: English ability, knowledge of AI planning techniques, and years of education.

It is not surprising that English ability and knowledge of AI planning techniques reduced training time. Better English proficiency should increase reading speed, and knowledge of AI planning techniques should make Diligent's plan representation easier to understand.

In contrast, the correlation with years of education was unexpected. It is unclear why more education should increase training time. Perhaps, more experienced subjects study more carefully.

While the training times had a large variance, the group that used demonstrations and experiments (EC_1) had a larger mean training time than the other groups. This might suggest that group EC_1 received better training. However, subjects followed detailed instructions during training, and the instructions for groups EC_1 and EC_2 were almost identical. The difference between EC_1 and EC_2 can be explained by the subjects in EC_1 having the worst English proficiency and the most education.

7.6.4 Discussion of Logical Edits

When comparing groups EC_1 and EC_2, the results suggest that experiments reduce the number of logical edits. Group EC_1 requires fewer edits than EC_2 for both procedures, but the difference is greater for the more complex Procedure 1. The number of edits for EC_1 remains fairly constant, while EC_2 requires many more edits on Procedure 1. This increase in EC_2's edits on Procedure 1 probably results from Diligent's operator creation heuristics creating more preconditions. Group EC_1's more constant number of edits suggests that experiments can help remove unnecessary preconditions.

When comparing groups EC_2 and EC_3, the results suggest that demonstrations help on simpler procedures. Group EC_2 requires fewer edits than EC_3 for the simpler procedure, but requires the same number of edits on the more complex procedure.

The difference between groups EC_2 and EC_3 does not seem to be influenced by the fact that subjects in EC_3 had to type in attribute values. For group EC_3, entering or changing an attribute value was only counted as one edit. Additionally, most attribute values were one word and spelling errors did not appear to be a problem.

This result for groups EC_2 and EC_3 seems to be a tradeoff between the benefits of demonstrations and Diligent's bias towards creating unnecessary preconditions.[19] Demonstrating a step saves edits because it takes one edit and identifies the step, its preconditions, and its state changes. It appears that subjects who demonstrated (EC_2) spent their time removing unnecessary preconditions, while those who used an editor (EC_3) spent their time adding missing preconditions and state changes.

7.6.5 Discussion of Errors in Identifying Steps

An initial concern was that the procedures were too easy, but it turns out that they were too difficult. The procedures were meant to be challenging but not to the point where some subjects had difficulty figuring out which steps to perform. For this reason, the differences between groups in identifying steps were unexpected.

Because Diligent's heuristics for learning operators assume correct demonstrations, mistakes in identifying steps probably affected the groups that learned preconditions from demonstrations (EC_1 and EC_2) more severely than the group that specified preconditions with an editor (EC_3).

[19]It is easier for Diligent to remove unnecessary preconditions than for it to identify missing preconditions (Chapter 5).

On Procedure 1, the groups that demonstrate (EC_1 and EC_2) have fewer errors. The difference between group EC_3 and the other groups may be influenced by several factors. One potential factor is the abstraction of Procedure 1's description. However, the descriptions of both procedures do not seem very different. Procedure 1's description is simply less explicit in describing the ordering of the steps. Another potential factor is the complexity of Procedure 1, which has many more step relationships than the other procedure. For Procedure 1, maybe using only an editor (EC_3) is more cognitively challenging than demonstrating the steps (EC_1 and EC_2). Procedure complexity seems a more likely explanation than the description's abstraction, but this is area for further study.

Because Diligent assumes that authors know a procedure's steps, group EC_3's performance suggests performing a future study that eliminates the influence of invalid steps. This could be done by giving the subjects a valid sequence of steps. The subjects would then have to determine the causal links and ordering constraints between the steps, which is what Diligent is designed to learn.

The differences between groups on Procedure 2 are minor. However, groups EC_1 and EC_2 should have had similar values because both groups use demonstrations. The subjects in EC_2 did a better job of demonstrating the procedure because that group produced a higher percentage of procedures that would have worked. Group EC_1's problems demonstrating might have counteracted the benefits of Diligent's experiments because group EC_1's procedures had more errors after demonstrations.

7.6.6 Discussion of Errors of Omission

Diligent's heuristics are designed to avoid errors of omission. If a procedure is demonstrated correctly, there should be few errors of omission.

In Procedure 1, mean errors for all groups are better than they appear because 4 of those errors are step specific control preconditions that are not required by the environment and, thus, are not learned by Diligent.[20]

In Procedure 1, group EC_3 is much worse than the other groups. However, the subjects in group EC_3 did a much worse job of identify the procedure's steps. This poor identification of steps might have exaggerated the differences between groups.

Because Procedure 2 has fewer step relationships than Procedure 1, the number of errors of omission should be lower. This what was found for all groups but EC_1. Although

[20]In Procedure 1, two valves should be opened when their alarm lights are illuminated and should be shut when their lights turn off. The environment requires the valves to opened because of internal pressure.

the subjects in group EC_1 did a poor job of identifying the procedure's steps, they still did as well as the subjects in group EC_3, who did a better job of identifying the steps.

One striking result is the large number of errors of omission for the group that used only an editor (EC_3). This was not entirely unexpected because subjects in EC_3 have to explicitly specify all steps, their preconditions, and their state changes.

It seems unlikely that group EC_3's large numbers of errors result from the group's being required to type in attribute values. Most attribute values were one word, and there were not that many preconditions and state changes. Perhaps spelling errors caused problems? But when examining the subjects' procedures, spelling errors were not an issue, and when asked, several subjects indicated that spelling errors were not a problem.

When comparing groups EC_1 and EC_2, experiments did not reduce the number of errors of omission. This was expected because Diligent has a bias towards errors of commission.

When comparing groups EC_2 and EC_3, demonstrations reduced the number of errors of omission in the complex procedure, but appeared to have little benefit in the simple procedure. This suggests that, when using an editor (EC_3), the number of step relationships is correlated with the number of errors of omission.

7.6.7 Discussion of Errors of Commission

When comparing groups EC_1 and EC_2, the results suggest that experiments reduce the errors of commission. The benefits of experiments are greater in the more complex procedure.

When comparing groups EC_2 and EC_3, the results suggest that demonstrations result in more errors of commission. More errors were committed in the complex procedure where Diligent's heuristics created more unnecessary preconditions.

7.6.8 Discussion of Total Errors

When comparing groups EC_1 and EC_2, the results suggest experiments will only reduce the number of errors on complex procedures. If both groups had equally good demonstrations for the second procedure, then experiments might also have shown a benefit on the simple procedure.

When comparing groups EC_2 and EC_3, the results suggest demonstrations only reduce the number of errors on complex procedures. This reduction occurred even though the groups had a similar number of logic edits.

7.6.9 Discussion of Total Required Effort

The total required effort is a measure of amount of work required to produce a correct plan. The required effort includes the work that has been done as well as the work that needs to be done. Work is measured by logical edits, and future work is estimated by total errors.

On Procedure 2, the increase in total required effort after testing suggests that total errors underestimates the additional work that needs to be done. Because domain experts might be better able to identify errors, it is unclear whether total errors would underestimate the future work for domain experts.

When comparing groups EC_1 and EC_2, the results suggest that experiments reduce the total required effort and have a greater effect on more complex procedures.

When comparing groups EC_2 and EC_3, the results suggest that demonstrations reduce the total required effort and have a greater effect on more complex procedures.

7.6.10 Discussion of Time Spent Authoring

There was a 30 minute time limit placed on each procedure. It was assumed that the subjects in group EC_1 would be able to author each procedure in approximately 15 minutes. However, the results indicate that 30 minutes was not enough time. This probably results from the subjects being unfamiliar with the domain, and the fact that the subjects had determine which steps to perform. Oftentimes, subjects would spent around 10 minutes studying the procedure description before starting to author.

In Procedure 2, the last 6 subjects started testing much earlier than most previous subjects. It is unclear why this so. Maybe there was a change in the experimental setup. Perhaps, the subjects were able to understand the directions better because they had a better description for the first procedure. However, the statistics involving logical edits and errors don't appear different for these subjects.

If subjects were given more time, they might have produced better procedures. This means that they may have made fewer errors and performed more logical edits.

The fact that subjects in group EC_3 had to type in attribute values may have increased the times for this group slightly. However, relatively little typing was needed, and subjects appeared to make few spelling mistakes. Therefore, the impact of typing is probably relatively minor.

Because of the time limit placed on subjects, we cannot draw any conclusions about whether demonstrations or experiments reduce the amount of time spent authoring.

7.6.11 Discussion of Subjective Impressions

The last thing the subjects did during the study was fill out a questionnaire about their subjective impressions of Diligent. The subjective impressions focus on aspects of the user interface. The impressions provide some indication of the usability of the three versions of Diligent. The impressions also indicate how subjects perceived various features.

There is relatively little data and a lot of variation between subjects.

A number of factors probably influenced the subjects' impressions. The impressions likely reflect both the training and the evaluation. The subjects' ratings may reflect ambiguities in the menus and difficulties with the environment's graphical interface. For example, some subject's who experienced software problems gave lower ratings. Additionally, the difficulty of the procedures being authored was probably also a factor.

All groups indicated that they liked the system somewhat, and the group that experimented (EC_1) liked it a little better than others. Unfortunately, this question is ambiguous because it does not indicate whether it is asking about only Diligent or about all the software used for authoring (e.g. environment's graphical interface).

The subjects found the system a little difficult to use. However, this may reflect the difficult procedures being authored. Subjects that experimented (EC_1) found it a little easier than subjects who only demonstrated (EC_2). Subjects who only used the editor (EC_3) found it the most difficult. This pattern was expected because using only an editor is difficult.

All groups indicated that it was somewhat easy to specify steps, preconditions and state changes. This is very desirable. *This is an indication that all three systems are reasonable and that editor only version (EC_3) is not a straw man.*

The ratings for the ease in identifying "how operators influence preconditions and state changes" are confusing. It is unclear why the different groups are so different. The group that experimented (EC_1) found it easier than the group that only demonstrated (EC_2). Maybe this is a reflection of how experiments improve preconditions. The group that only used the editor (EC_3) had the most difficulty. Maybe this indicates that representing preconditions and state changes with operators is more difficult when using an editor. This might also indicate that it is harder to determine the correct preconditions and state changes when using an editor.

Subjects found it somewhat easy to demonstrate. Although the groups that demonstrated have slightly different means, both groups used the same techniques for demonstrating. It is surprising that the rating is this high given the problems during the first half of the study when a memory leak caused the environment's graphical interface to be unresponsive and slow.

The subjects also found the ability to provide additional demonstrations of a procedure somewhat useful. Before starting the evaluation, it was assumed that few subjects would need this capability. It is unclear whether this rating reflects the training or the subjects' difficulty in identifying a procedure's steps.

Subjects only somewhat liked having Diligent perform autonomous experiments. Some subjects seemed to feel that experimenting was a strange feature and were not sure what to think about it. Subject 12 indicated a strong dislike of experiments even though the subject felt that experiments were useful and quick.

Subjects felt that experiments were quick enough. This provides support for the arguments in Chapter 6 that the experimentation approach has a reasonable run-time complexity.

There was strong support for the correct conclusion that experiments save work, but subjects had a only moderate belief that experiments would have caught errors that they would have missed. This contradicts the data for errors of commission on Procedure 1. The data for Procedure 1 suggest that experiments prevented many errors in the final procedure.

7.7 Reviewing the Claims

Now that we have discussed the results, we will look at how well the results support the hypotheses presented in Section 7.1.

Because there were few subjects, statistical significance was rarely achieved. Moreover, the probabilities should be viewed with a little skepticism because they are too sensitive to individual data points. However, the results are still valuable because they indicate patterns and trends.

The claims compare group EC_1 against EC_2 and group EC_2 against EC_3. However, group EC_2 often has the intermediate value. This means that statistically significant differences between EC_1 and EC_3 are not used to justify the claims.

The various claims are addressed by data in the following sections. Claims 1 and 2 are addressed by the data for logical edits (Section 7.5.3), which is discussed in Section 7.6.4. Claims 3 and 4 are addressed by the data for total errors (Section 7.5.4.4), which is discussed in Section 7.6.8. Claims 5 and 6 are addressed by the data for total required effort (Section 7.5.5), which is discussed in Section 7.6.9. Claims 7 and 8 are addressed by the data for the time spent authoring (Section 7.5.6), which is discussed in Section 7.6.10.

- *Claim 1: Subjects require less work to create a procedure when using demonstrations and experiments than when using only demonstrations.*

  This claim is supported. However, there appears to be less benefit on simpler procedures.

- *Claim 2: Subjects require less work to create a procedure when using only demonstrations than when using only an editor.*

  This claim is partially supported. On complicated procedures, there does not appear to be a difference between using demonstrations or an editor. However, demonstrations appear to provide an advantage on simpler procedures.

- *Claim 3: Using demonstrations and experiments results in fewer errors than when using only demonstrations.*

  This claim is partially supported. On complicated procedures, experiments appear to be beneficial. However, experiments do not appear to be that useful on simpler procedures.

  One problem with this claim is that the subjects who experimented did a poor job of demonstrating the simpler procedure. This caused the group that experimented to have more errors of omission than the group that didn't experiment.

- *Claim 4: Using only demonstrations results in fewer errors than when using only an editor.*

  This claim has weak partial support. On the complicated procedure, demonstrations seemed to help, but demonstrations did not appear to have much effect on the simpler procedure.

- *Claim 5: Subjects require less work to create a correct procedure when using demonstrations and experiments than when using only demonstrations.*

  This claim is supported. However, the benefits of experiments are less on simpler procedures.

- *Claim 6: Subjects require less work to create a correct procedure when using only demonstrations than when using only an editor.*

This claim is supported. However, the benefits of demonstrations are less on simpler procedures.

- *Claim 7: Subjects can author in less time using demonstrations and experiments than when using only demonstrations.*

  The data are inconclusive. The time spent authoring indicates only a small difference, and most subjects appeared to have run out of time before they were finished.

- *Claim 8: Subjects can author in less time using only demonstrations than when using only an editor.*

  The data are inconclusive. The time spent authoring indicates only a small difference, and most subjects appeared to have run out of time before they were finished.

Dependent Variable	**Relation**	**Holds on**		**Direction of increased difference**
		Simple	**Complex**	
Edits	$EC_1 > EC_2$	Yes	Yes	complex
	$EC_2 > EC_3$	Yes	No	simple
Errors	$EC_1 > EC_2$	No	Yes	complex
	$EC_2 > EC_3$	No	Yes	complex
Total Effort	$EC_1 > EC_2$	Yes	Yes	complex
	$EC_2 > EC_3$	Yes	Yes	complex
Time	$EC_1 > EC_2$	-	-	
	$EC_2 > EC_3$	-	-	

Table 7.16: Summary of Results

These results are summarized in Table 7.16. The relations compare the groups that experimented (EC_1), only demonstrated (EC_2) and only used an editor (EC_3). The relation $A > B$ means that A does better than B. The results indicate that experiments help more on complex procedures. An interesting result is that subjects who only demonstrated on the complex procedure had as many edits as those who used the editor, but the subjects who demonstrated produced fewer errors. Neither experiments or demonstrations appeared to reduce errors in simple procedures, but they did reduce errors in complex procedures. The total effort required to produce a correct procedure includes both edits and errors. The total required effort was reduced by both experiments and demonstrations. Because of time restrictions, no conclusions could be made about time spent authoring.

In hindsight, the patterns found in this study appear reasonable, and it seems likely that the patterns would be maintained if the test subjects were domain experts rather than graduate students.

7.8 Observations

During the study, a few miscellaneous issues were observed.

- After subjects finished the evaluation, they were given a demonstration of Diligent. One remark that was heard several times was that they hadn't realized how to use Diligent effectively. One reason for this is that Diligent has a very unusual user interface. The subjects indicated that they would have liked to have seen a demonstration of Diligent at the start of training. However, demonstrating the system separately for each subject would have introduced a great deal of variation in the training of subjects. One way to deal with this issue, when testing systems with unusual types of user interfaces, is to play a video that illustrates how to use the system.

- There is a tradeoff between asking test subjects to perform simple versus complicated tasks. A simple task is more likely to get statistically significant results, but if a task is too simple, the results may be trivial because the task is too much of a toy problem.

  The tradeoff is relevant to this study because Diligent focuses more on understanding demonstrations than on the usability of its user interface. By not telling subjects a valid sequence of steps, the authoring task was made more challenging, but one of Diligent's assumptions was violated. The challenging procedures introduced more variability into the study and placed more emphasis on the user interface.

  Before the study, some user interface features were thought to be insurance rather than necessities (e.g. the ability to delete steps). However, subjects used these features quite often. Additional features that were deemed unnecessary were sometimes requested by subjects (e.g. a dynamically updated graph of a procedure).

  A related issue is the amount of flexibility allowed by the user interface. The usability testing identified the need to use forcing functions to prevent very undesirable behavior. The formal evaluation also indicated a need to disable features that are irrelevant to the task. For example, although the ability to create hierarchical procedures was not discussed during training, one subject created a hierarchical procedure.

7.9 Summary

This chapter discussed an empirical evaluation of Diligent. Instead of focusing on how well Diligent could understand demonstrations, the study focused on how Diligent's techniques help a human author.

The study had a between-subjects design where the subjects were divided into three groups. The subjects in a given group had similar training and used the same version of Diligent. After approximately two hours of training, the subjects authored two procedures. One of the procedures could be considered more complicated because it is a little longer and has many more step relationships. Finally, subjects gave their impressions of Diligent in a post-test.

The differences between the three versions of Diligent involved demonstrations and experiments. One version supported both demonstrations and experiments, while another version used demonstrations but did not allow experiments. A third version provided an editor and did not support demonstrations. The user interface for the three versions was as similar as possible. The versions that used demonstrations were basically identical. The version that only provided an editor differed from the others in how steps were added to a procedure and in how preconditions and state changes were specified. The results of the post-test suggest that subjects felt that the editor only version was reasonable and fair.

The study identified benefits of using demonstrations and experiments. Using experiments and demonstrations appeared to be better than just using demonstrations, and using demonstrations without experiments appeared to be better than using only an editor. The differences between the groups appear greater on complex procedures. Experiments reduced the number of edits that subjects performed, while demonstrations only appeared to reduce the number of edits in simpler procedures. Although neither experiments nor demonstrations appear reduce errors in simple procedures, both experiments and demonstrations appear to reduce errors in complicated procedures. When considering both edits and errors, both experiments and demonstrations appear beneficial for both simple and complex procedures. Because of time restrictions, the study could not determine how experiments and demonstrations influenced the time spent authoring.

The responses to the post-test suggest that Diligent's experimentation approach is acceptably fast on procedures of 6 to 8 steps, which is approximately the expected size of non-hierarchical procedures.

Chapter 8

Analysis and Future Work

In Chapter 3, we discussed Diligent at high level. The subsequent chapters then focused on individual topics, such as processing demonstrations, learning operators and experimenting. This background enables us to have a more unified discussion of Diligent, including its limitations and potential extensions.

This chapter is organized in the following manner. We will first discuss how Diligent's methods address the problem of understanding demonstrations by discussing several perspectives for viewing demonstrations. We will then talk about assumptions and how easily they can be relaxed. Afterwards, we talk about limitations and potential extensions.

8.1 Perspectives for Understanding Demonstrations

One way that Diligent addresses the problem of understanding demonstrations is by viewing a demonstration from multiple perspectives. Each perspective asks a different question, and by focusing on each question, demonstrations can be better understood.

One could view Diligent as a set of methods that address the following four questions.

When should a step be performed? Under what conditions should a step be performed in order to achieve the procedure's goals? This perspective deals with identifying knowledge for *controlling* when steps are performed. In contrast, the other perspectives deal with how the environment *functions* independently of the procedure's goals. Diligent methods address this question in the following ways.

Knowing when to perform a step requires knowledge of the procedure's goal conditions. Diligent proposes a set of goal conditions to the instructor that contain the final values of attributes that change value during the procedure. When a procedure's goal conditions are satisfied, the procedure terminates because no more steps are necessary.

Diligent computes when to perform steps by analytically deriving step relationships (i.e. causal links and ordering constraints) between a procedure's steps. Later, when performing the procedure, the step relationships identify which steps are currently applicable.

Some preconditions of steps come from operators, which reflect how the environment functions independent of the procedure; but other preconditions are associated with individual steps. Consider sensing actions (e.g. examining a gauge), which gather information from the environment without changing its state. Because the environment may allow a sensing action to be performed anytime, a sensing action's operator might not have any preconditions. For this reason, a sensing action's preconditions are associated with its step. Sensing actions need preconditions to ensure that they are performed in the proper place within a procedure. By default, a sensing action's preconditions contain the attributes that have changed value before the sensing action during the demonstration.

What pre-state conditions are common when a given state change is seen? This is an instance of the standard concept learning question.

Diligent addresses this issue with its version space algorithm for learning operator preconditions. An advantage of this approach is that Diligent can learn from both positive and negative examples.

What is different when different state changes are seen? The question deals with comparing the preconditions of effects that produce different state changes.

This perspective is used when creating an effect for an operator that already has an effect. When identifying the heuristic preconditions (h-rep), Diligent uses the h-rep of an existing effect but adjusts it with the current action-example's pre-state. The new h-rep also contains conditions in the current action-example's pre-state that are different than ones in the most similar earlier action-example. (Only the current example is positive for the new effect, while earlier examples are negative.)

Diligent also compares effects with different state changes. When the h-rep of one effect cannot correctly reject a negative example, conditions from the example's pre-state may be compared to the preconditions of other effects for which the example is positive. If a one condition match is found, a condition is added to the h-rep.

Why isn't a step earlier? The instructor probably has reasons for demonstrating steps in a given order. One reason is that the state changes of some earlier steps are likely to be preconditions of some later steps. Diligent is novel in how it emphasizes this question.

Diligent has a couple heuristics that deal with this perspective: focus on attributes that change value, and earlier steps are likely to establish preconditions of later steps.

Diligent uses this perspective when creating an operator's first effect. The initial h-rep contains attributes that have changed value during the current demonstration.

A similar approach is used to create preconditions for sensing actions.

Diligent's experiments also focus on this perspective by skipping a step and observing how later steps are impacted.

8.2 Assumptions

This section discusses the assumptions used by Diligent and the difficulty in relaxing them.

8.2.1 Easier to Relax

Relaxing the following assumptions appears to be relatively easy.

No attributes are added or removed from the environment. In Diligent's domains, the number of attributes in the environment's state is constant. This can be reasonable in tutorial domain because students might get confused if attributes were being added or removed. In any case, there hasn't been a need to relax this assumption when using Diligent.

Previous work by Wang [Wan96a] has relaxed this assumption, and her technique could be incorporated into Diligent.

However, there are a couple special cases where relaxing this assumption would be more difficult.

- If the domain is under development, then new attributes could be added to the environment and existing attributes could be used differently. If new attributes don't affect existing procedures, then this might not be a major problem, but if the new attributes do affect existing procedures, then it may be difficult to use previously learned knowledge.
- The domain is so large that agents (e.g. Diligent) are given a limited view the domain. For example, agents in each simulated room might see different sets

of attributes. If an agent's current view did not include all relevant attributes, then there could be difficulties. (This issue is discussed below under relaxing the assumption that all relevant attributes are visible.)

No generalized conditions. When Diligent uses a condition, the condition always refers to a specific attribute and a specific value.[1] An alternative would have been to introduce variables into preconditions and state changes. Operators containing variables could then apply to multiple objects of the same class. Diligent's approach was used for three reasons: there is relatively little input data; the environment's lack of structure hides relationships between objects and attributes; and many objects (e.g. switches) have idiosyncratic behavior. As an example of idiosyncratic behavior, consider two switches; one switch may turn on some lights, while another switch may start the motor.

If many objects of the class have similar behavior, then introducing variables into preconditions and state changes could allow more generally applicable operators. This type of approach is also looked at by OBSERVER [Wan96c].

Qualitative attribute values. Attributes are assumed to have only a few discrete values rather than continuous or numeric values. For example, a temperature sensor might only have the values ok and too-hot.

Qualitative attribute values have several advantages. They are easy to use with machine learning algorithms, and they may provide descriptions that humans find conceptually easy to understand (e.g. too-hot).

However, sometimes qualitative attributes are not appropriate. It might be difficult to identify meaningful qualitative values, or there might be a large number of values that are associated with qualitatively different behavior. Moreover, sometimes it may be important for people know numeric values or know relations between values (e.g. height < 5).

Whether qualitative or quantitative attribute values are used, an important issue is what is the most effective authoring method. Determining this may involve considering both the ease of authoring and the quality of student remediation.

Qualitative attribute values are required by Diligent's learning algorithms, but this restriction could be overcome by associating a range of numeric values with a single

[1]An exception is conditions involving mental attributes. These conditions indicate that their value is unimportant.

qualitative value. This technique also could be used with numeric formulas or conditions involving relations other than equality (e.g. temp < 5) [Wd90]. Providing the ability to assign numeric ranges to a qualitative value appears easy.

However, it is unclear whether Diligent would ever get enough data to automatically generate quantitative boundaries (e.g. numeric formulas) that specify a qualitative attribute value (e.g. too-hot).

Conjunctive preconditions and goals. Although a step might produce state changes from several of its operator's effects, Diligent assumes that the preconditions of a step are conjunctive. Diligent also assumes that a procedure terminates when its conjunctive goal conditions are met.

Allowing disjunctive preconditions raises two issues. First, learning disjunctive preconditions may take more data than learning conjunctive preconditions. Second, the system would need to determine which disjuncts correspond to each step. This should not pose a problem if the preconditions are very refined, but could be problematic when the preconditions are less refined.

Disjunctive preconditions would probably require more interaction with the instructor. Presently, disjunctions can only be detected when the version space collapses.

Disjunctive goal conditions seem more problematic. Specifying disjunctive goal conditions does not seem difficult, but would require at least one demonstration of each disjunct. However, using a subprocedure with disjunctive goals appears more difficult. If a subprocedure's abstract step could have multiple distinct post-states, then each post-state might require a different sequence of subsequent steps in the parent procedure.

Instructor correctly demonstrates procedures. If the instructor doesn't correctly demonstrate a procedure, the procedure's path will not produce a correct plan. Moreover, Diligent's heuristics assume that there is a good reason for the sequencing of a procedure's steps.

Correcting a path poses no problem, but an invalid sequencing of steps might lead to worse heuristic preconditions. Although experiments might help, correcting the preconditions might require that the instructor provide more training data. An aspect of this problem is that Diligent's learning algorithms can more easily remove unnecessary hypothesized preconditions than identify missing ones.

8.2.2 Harder to Relax

Relaxing the following assumptions appears to be relatively difficult. Relaxing most of these assumptions does not appear particularly important.

One action at a time. Diligent assumes that only one action takes place at a time. This helps in identifying preconditions and state changes. It also helps in determining the sequence of a path's steps.

If the instructor were able to perform several actions simultaneously and if these actions could have been performed sequentially, the action-examples of these actions might be misleading because a post-state could contain the results of several actions. To handle this situation, Diligent could either use a more robust operator learning algorithm or delay learning until it has had a chance to replay the demonstration with the actions separated by time.

Deterministic actions. In a given pre-state, Diligent needs to know which state changes will be caused by a given action. Actions appear non-deterministic when a relevant environment attribute is not seen [She94].

Sometimes an action appears non-deterministic when it needs to be repeated several times. An example from the HPAC domain is a dipstick which needs to be selected several times when being extracted. When the dipstick is in its intermediate position, Diligent cannot tell whether selecting it will move dipstick in or out of its hole. An action, like selecting the dipstick, that is repeated several times could be modeled by considering the state changes after last action is performed. Understanding repeated sequences of actions is an important issue for robotic programming by demonstration systems [Hei93, FMD+96].

Handling non-determinism in actions that can be repeated until they produce the desired result appears to be easy and important, but it is unclear how the system should handle other cases of non-determinism.

One problem with non-deterministic actions is handling non-determinism during experiments. How does the system detect non-determinism? Perhaps, experiments could be repeated several times.

Can tell when an action begins and ends. It is implicit in Diligent's interface with environment that action-examples will identify an action's pre-state and post-state. This knowledge is required to identify preconditions and state changes.

Relaxing this assumption in general appears fairly difficult and may not be very important.

However, delayed state changes (or delayed effects) could important. A delayed state change occurs when an action is finished but a future state change has not yet happened. For example, a copy machine may not finish warming up for a minute after it is started. Consider the following cases:

- The delayed state change happens before the next action. In this case, the system could notice the change and associate it with the previous action.
- The delayed state happens after subsequent unrelated actions and changes to the state, but does not happen during a later action. In this case, the system might look for the last action that changed the state change's attribute. For example, when starting a copy machine, the machine may take one minute to warm up before it is ready. In this case, the state of the machine might go from off to warming-up when the machine is started and after one minute to ready. A system might then infer that starting the machine eventually caused it to become ready.
- The delayed state happens after subsequent unrelated actions and changes to the state and happens during a later action. In this case, the environment would appear non-deterministic. It is unclear how this should be handled. Perhaps, a system could detect this if enough training data were available.

Can see all relevant attributes. An attribute is considered relevant when it is needed for teaching or for operators to appear deterministic. Besides non-determinism, which we've discussed, this assumption impacts teaching. An attribute is useless for teaching when neither Diligent nor an automated tutor can see it.

Relaxing this assumption appears difficult. Maybe missing attributes could be represented by mental attributes, but it is unclear how well this would work.

Noise-free sensors. Diligent assumes that the data it gets from the environment contains no errors. This is important because little data is received, and the lack of data would make recovering from errors more difficult.

A method for relaxing this assumption would be to replay demonstrations and repeat experiments. Multiple action-examples for each step could then be compared. Of course, this approach would take more time.

Can see all actions. Diligent's ability to record demonstrations depends on its ability to observe all actions performed in the environment.

Relaxing this assumption appears fairly difficult.

No exogenous events Exogenous events are things that happen to the simulated domain that are not caused by the user or by the authoring tool (e.g. Diligent). For example, exogenous events include actions performed by other agents or special events in the simulated world (e.g. a fire starting in the engine room).

If the authoring tool knew that an exogenous event was an exogenous event, then it should not be that difficult to model it. Otherwise, handling exogenous events is similar to not being able to see all actions.

Partially ordered procedures. If a procedure has only one valid sequence of steps, then Diligent's experiments might not learn anything useful. Experiments attempt to produce new action-examples for refining the preconditions of desired state changes. In an experiment on a totally ordered procedure, all examples might be negative. These negative examples might identify necessary preconditions, but the examples would not remove any unnecessary preconditions.

Relaxing this assumption appears difficult.

Modular procedures. Diligent performs fewer actions in experiments when large procedures are divided into modular subprocedures.

It's unclear how to relax this assumption. Perhaps, a system could perform experiments when the instructor was not present. However, if the instructor is present, the number of actions performed might be reduced by examining operator preconditions and only performing experiments likely to refine preconditions.

Non-interleaved plans. *Interleaved* plans [RN95][2] interleave the performance of the steps of two subprocedures. When the instructor uses subprocedures in demonstrations, he uses the subprocedures sequentially. This makes Diligent incapable of learning a procedure whose subprocedures can only achieve their goals by interleaving their steps.

Relaxing this assumption appears difficult.

[2]These types of plans have also been called non-linear.

Diligent can reset the environment. Diligent's techniques assume that it can reset the state of the environment.

Although some of the ideas that Diligent uses to understand demonstrations might be useful, Diligent's algorithms are probably inappropriate for an agent that cannot reset the state of the environment.

8.3 Limitations

8.3.1 Coordinated Simultaneous Actions

Besides the limitations inherent in a direct manipulation interface [Coh92], Diligent's use of a single manipulation device (i.e. mouse) caused problems in the HPAC domain. In particular, the HPAC's Temperature Monitor requires the user to perform pairs of actions simultaneously: the read reset and trip temperature buttons need to be depressed simultaneously to view the temperature at which the currently selected sensor will illuminate an alarm light. People can do this with two hands, but it is unclear how to do this with only one mouse.

A related issue is when to consider similar types of actions finished. In the above example, the temperature displayed on the gauge disappears when the buttons are released. Thus, Diligent would not even see the temperature because its action-examples treat depressing and releasing a button as an atomic action and hide intermediate states. An alternative is having separate action-examples (and steps) for pressing and releasing a button. However, this alternative is likely to irritate humans. This raises the question of how to uniformly process a given type of action (e.g. pressing buttons).[3]

Extending Diligent to handle coordinated simultaneous actions might require modeling a set of simultaneous actions with a single operator.

8.3.2 When Pre-State and Post-State Values are Independent

Diligent has problems learning operators when an attribute's post-state value does not depend on its pre-state value. When this happens, the attribute may have its value reset but to the same value as in the pre-state. The problem is distinguishing between situations where the attribute's value is and is not reset.

[3]Because Diligent gets action-examples from the environment (Section 3.1.3), it's the environment's responsibility to make decisions on when to create action-examples.

This indeterminism reduces the number of positive examples available for learning. If an attribute has its value reset to its pre-state value, the example cannot be classified as positive because Diligent cannot tell that it was reset. If the value wasn't reset, then some necessary preconditions were unsatisfied; in this case, treating the example as positive could eliminate necessary preconditions and cause the version space to collapse.

This problem is worse for attributes that take only two values. If both pre-state values are equally likely and do not affect the post-state value, then one half of the "real" positive examples cannot be used.

This situation is illustrated by an example from the HPAC domain. In figure 8.1, the attribute CurrentValveIsOpen indicates whether the valve under the handle that manipulates valves is open. If the handle is moved to valve2, CurrentValveIsOpen changes its value without appearing to change. Therefore, the attribute is not listed in the example's delta-state (delta-state 1). In contrast, if the handle is to valve3, the attribute's value changes from true to false (delta-state 2).

Action-example:
- **Pre-state**:
  - *(CurrentValveIsOpen true)*
  - (valve1 open)
  - (valve2 open)
  - (valve3 shut)
  - (HandleOn valve1)

- **Delta-state 1**: (when moving to valve2)
  - (HandleOn valve2)

- **Delta-state 2**: (when moving to valve3)
  - *(CurrentValveIsOpen false)*
  - (HandleOn valve3)

Figure 8.1: An Attribute whose Post-State is Independent of its Pre-State

In this case, the problem results from an attribute (i.e. CurrentValveIsOpen) that contains redundant information. Instead, this fact could have been inferred from other observable attributes.[4] This suggests that a program that learns preconditions can have

[4]The STEVE tutor [RJ99] uses an attribute like CurrentValveIsOpen for determining when a handle has been turned. When evaluating Diligent, the attribute was filtered out so that neither students nor Diligent

problems when the simulation that controls the environment uses certain modeling techniques, but this topic is beyond our present scope.

One way to deal with this problem is to classify an action-example as positive for only one effect. Thus, even if an attribute didn't appear to change value, the action-example could be classified as positive because of changes in other attributes. Of course, this approach only works when effects contain multiple state changes.

Unfortunately, this approach must deal with misclassified examples. Because an action-example is a positive example of only one effect, multiple effects could change the same attribute. As a result, it could be difficult to determine which effect has the positive example. Later, when effects are more refined, it might be discovered than an action-example was misclassified as a positive example of a given effect. After detecting a misclassification, there is the overhead of recalculating two effects: the effect with the false positive and the effect with the false negative. Even worse, the scope of the recalculation is unclear because recomputing one effect may identify a misclassification with a third effect.

An algorithm of this type was implemented for Diligent. The algorithm worked well in the HPAC domain, but was removed out of concern about the worst case performance in domains where misclassifications are likely.[5]

The type of operators we've just described are called *relational* (or sometimes rewrite rules). For relational operators, the entire pre-state as a whole is transformed into the post-state. An example of a relational operator is a mathematical transformation such as performing symbolic integration on an integral. Relational operators have been discussed in work by Langley [Lan80] and by Porter and Kibler [PK86]. Their approaches, however, use domain dependent state transformation rules.

8.3.3 Transitive Dependencies

Diligent's experimentation approach may not work well when a procedure's set of steps is totally ordered. A set of steps is totally ordered when there is only one valid order for performing the steps. The problem is that skipping a step early in the procedure impacts each of the later steps.

The problem could be classified as involving transitive dependencies. A step Z has a *transitive dependency* on an earlier step X when Z depends on intermediate step Y and

saw it. In this case, the attribute's change in value could have been successfully modeled with disjunctive preconditions.

[5]Diligent's user interface provides some support for attributes like CurrentValveIsOpen. Instructors can edit preconditions and can filter out unwanted attributes so that they do not appear in plans.

step Y depends on step X. In other words, there are causal links from X to Y and from Y to Z. Skipping step X in an experiment interferes with Y, and anything that interferes with Y also interferes with Z. Thus, Diligent cannot determine whether Z has a causal link with X or whether Z is indirectly dependent on X through Y's causal link with X.

Other than experimenting with multiple paths, Diligent's experimentation technique does not address this problem. This appears to be a general problem of systems, like Diligent, that learn procedure independent knowledge (e.g. operators) by observing sequences of steps. In contrast, it may not be a problem for systems that learn when to perform steps (i.e. learn control knowledge) without understanding the dependencies between steps (i.e. causal links).

8.4 Extensions

In this section, we will discuss extensions that could enhance Diligent. We will first discuss extensions to the procedural representation because they motivate some of the extensions to authoring. We will then finish the section by discussing extensions to learning and experimentation.

8.4.1 Procedural Representation

Every procedure has one or more paths, but only one path is actually used to generate a plan. If Diligent allowed multiple paths to be used for generating plans, then instructors could author a larger set of procedures. Multiple paths could support starting a procedure in a variety of initial states. Multiple paths could also support conditionally performing steps based on the state earlier in the procedure. The following sections discuss ways to use multiple paths.

8.4.1.1 Multiple Methods for Performing a Procedure

Originally, Diligent allowed the instructor to specify different orders of steps for performing a procedure. A different order of steps resulted in an additional path. This capability not only supported different initial states, but also allowed the relative order of some actions to be reversed in different paths. However, this capability was removed because of the problems described below and because none of the procedures authored with Diligent required this capability.

When a procedure has different paths for achieving its goals, the relative order of some actions in different paths might be reversed. If the paths are used to create a single plan, the plan could contain circular dependencies (i.e. step relationships) between its steps.

This actually happened with the paths shown in figure 8.2. In the figure, M2 represents moving the handle to the second valve, and S2 represents shutting the second valve. There were two demonstrations, and each created a different path. In one demonstration, the handle was initially moved to the second valve, while in the other demonstration, it was initially moved to the first valve.

path A:
move to second valve (M2) → shut second valve (S2) →
move to first valve (M1) → shut first valve (S1)
path B:
M1 → S1 → M2 → S2

Desired plan:
goto first or second valve → shut the valve →
goto the other valve → shut the other valve

The Problem is Step Relationships:
. . . → M2 → S2 → M1 → S1 → M2 → S2 → . . .

Figure 8.2: Incompatible Paths

If the identifiers used for the steps in one path are reused for the equivalent steps in the other path, the procedure will only contain four steps (i.e. S1, M1, S2 and M2). When the step relationships for the two paths are used in the same plan, there is a circular dependency between steps (i.e. the second valve needs to be shut before the first valve, and the first valve needs to be shut before the second valve.). Because of this circularity, the plan cannot be executed without violating some of the dependencies.

One approach is to create ordering constraints that favor one path over another. However, it was unclear which was the best method for doing this. An approach used by Instructo-Soar [HL95] involves asking the user which step to prefer when multiple steps are applicable. However, this approach was not used by Diligent because we were focusing on machine learning rather than on complex interaction with the instructor.

A different problem appears when equivalent steps in different paths use different identifiers. In this case, the plan would contain eight steps. The problem with the resulting plan is that the first step of each path removes the state changes of the first step in the other path. Thus, a system using the plan could indefinitely move the handle back and forth between the first and second valves without ever shutting either one.

One solution is associating each step with a distance from the goal state. If multiple steps are applicable, then the system could then choose the step that was closest to the goal state [PK86]. However, using this approach would have required us to use a non-standard plan representation.

Another solution is to create several plans, or methods, for the procedure. For our purposes, a *method* is a plan of the procedure. When a procedure is started, an automated tutor would select the appropriate method. If there was a student error or an unexpected problem, the tutor might recover by switching to another method.

8.4.1.2 Conditional Plans

Diligent cannot learn conditional plans [PS92, DHW94, RN95]. A *conditional* plan contains branch steps, and different sequences of later steps are performed based on a decision made at a branch step. A *branch step* looks at the current state and determines which subsequent steps to perform based on whether its preconditions are satisfied. Branch steps can be thought of as creating a mental attribute whose value is a precondition for the steps following it. Consider the procedure "If the light is on, press buttons B and C; otherwise, just shut valve D." In this instance, the branch step checks whether the light is on.

Diligent could produce conditional plans by having two paths for every branch step. One path would represent an unsatisfied branch condition, while the other path would represent a satisfied branch condition.

Some of the issues involved include

- If a procedure already has multiple paths, how to incorporate demonstrations of each branch in multiple paths? Otherwise, the instructor may have to demonstrate the steps in a branch multiple times.

- Identifying the conditions that control which branch is performed. One heuristic is using the pre-state differences between the demonstrations of the two branches.

8.4.1.3 Disjunctive Goal Conditions

Diligent assumes that a procedure has conjunctive goal conditions. However, disjunctive goals are sometimes desirable, especially in conditional plans. For example, a plan might have one goal state for successful execution and another for unsuccessful execution.

An important issue is how to handle subprocedures that have disjunctive goals. When inserting a subprocedure into a parent procedure, the parent needs to handle all the subprocedure's goal states. Furthermore, after adding a disjunct to a procedure's goals, any use of that procedure as a subprocedure may require updating each parent procedure.

8.4.2 Authoring

This section discusses extensions that could make authoring easier, especially if procedures or domains are complicated.

8.4.2.1 Additional Types of Demonstrations

Diligent supports two types of demonstrations (Section 4.2): one type adds steps to a procedure's plan, and the other type provides data for machine learning without adding steps to the plan. Only two types of demonstrations were needed because only simple procedures were needed by the portion of the HPAC domain that was implemented.

However, if the procedure representation were more complicated, then the following types of demonstrations might also be useful.

Alternative-step-order. This type of demonstration allows instructors to demonstrate a procedure's steps in a different order or from different initial states. These types of demonstrations would support more robust procedures and provide more data for learning.

This type of demonstration was implemented and then later removed. (Section 8.4.1.1 discusses some of the issues.)

Branch. This type of demonstration would support conditional plans. The demonstration would start at the branch step and perform a sequence of steps based on the branch conditions. Because a branch requires at least two alternative sequences of steps, the pre-state of each sequence could help identify the branch conditions.

Undesirable-action. This type of demonstration would teach control knowledge. The environment would be put in a desired state and an undesirable action performed. The system could then compare pre-states where the action should be avoided to the pre-states where the action is applicable. One issue is how to incorporate this knowledge into the plan.

Applicable-state. This is similar to undesired-action demonstrations, but in this case, performing the action in the pre-state is desirable. This type of demonstration would be useful for refining branch conditions and the preconditions of sensing actions.

8.4.2.2 Continuous/Parameterized Actions

Diligent supports actions where the only parameter associated with an action is the object selected by the instructor. However, successfully modeling some types of actions requires associating more parameters with the action. In the two domains used with Diligent, several actions have this property.

- There is a temperature gauge that shows the temperature associated with one of about a dozen sensors. The actual sensor shown is determined by the position of a rotary selector switch.
- The thrust of the ship's engines is determined by the position of a throttle.

Actions involving the selector switch and the throttle attempt to move the object into a desired position. These types of actions could be modeled by associating the desired position with the action.

8.4.2.3 Types of Mental Attributes

A mental attribute is an attribute that is stored internally by Diligent, or an automated tutor, and is not present in the environment. The type of information represented by a mental attribute is an important issue. At least three types of mental attributes seem reasonable.

- The attribute is global. It represents the agent's knowledge of the world independent of which step sets its value. An example from a medical domain is whether someone's throat is obstructed.
- The attribute is specific to sensing actions in one procedure. For example, a conditional plan may test a light and repair it if it doesn't work. Because the state of the device is uncertain, this may involve several sensing actions that test whether the light is turned on. The fact that the light turns on might be treated the same regardless of which step actually observed that the light was on.
- The attribute is specific to one step. Diligent supports this type of attribute. For example, in Chapter 4, a mental attribute was used to represent that an alarm light

was checked while the HPAC was in test mode. Later in the same procedure, another sensing action could have created a different mental attribute to store the result of checking the alarm light when the system was no longer in test mode.

Another issue is what to do with mental attributes that are created by reused subprocedures. If the same subprocedure is reused multiple times in the same procedure, how should Diligent distinguish between mental attributes that are created by the different abstract steps that represent this subprocedure?

8.4.2.4 Inferred Attributes

An *inferred attribute* represents an attribute whose value is inferred from the values of other attributes. Inferred attributes could help the instructor create fewer, more abstract attributes. Consider a subprocedure that checks four alarm lights. Instead of returning a mental attribute for each sensing action, the instructor might create a single mental attribute that indicates whether all four lights work.

Inferred attributes could also be used to create qualitative attributes that assign the state of the environment to one of several categories.[6]

One approach for creating inferred attributes is using Kelly's Personal Construct Psychology [Kel55], which has been used to acquire knowledge for expert systems [SG88, Boo85]. This approach is also known as a *repertory grid*. The basic idea is for the author to create an attribute that differentiates between several examples. As the author creates more attributes, he constructs a framework for viewing the domain.

8.4.3 Learning

We will first discuss some simpler extensions before discussing some more involved extensions.

8.4.3.1 Simple Extensions

There are a few simple ways that learning could be improved.

Negated preconditions. In contrast to Diligent's conditions, which specify the value an attribute must have, *negated conditions* specify the value an attribute cannot have. Negated conditions are occasionally useful as preconditions.

[6] Diligent assumes that attributes have qualitative values.

Negated conditions have already been used with version spaces by OBSERVER [Wan96c]. In OBSERVER, a negated precondition is added if a negative example's pre-state has an attribute that was missing from the environment in earlier positive examples. OBSERVER, however, will not detect that a negated precondition is required if the attribute is present in earlier positive examples.

With attributes that are constantly present in the environment, negated conditions are only needed if an attribute can take more than two values. Suppose that an attribute only takes values X and Y. If the value X was undesirable, the condition could simple specify that the value has to be Y.

Because Diligent's environment doesn't have attributes added or removed, Diligent couldn't use OBSERVER's approach for learning negated preconditions. However, negated preconditions could still be detected. A negated precondition might be needed if a specific attribute value is never present in positive examples, while two or more other values are present in positive examples.

Correlating attribute values between effects. Sometimes an attribute value is highly correlated with positive examples and poorly correlated with negative examples. Diligent could be extended to infer that these attribute values had a higher likelihood of being preconditions. If an attribute value gets a high enough likelihood, it could even be added to Diligent's heuristic preconditions (i.e. h-rep).

Disjunctive preconditions. Assuming that all relevant attributes are visible, a disjunctive precondition can be inferred when the version space collapses. Supporting disjunctive preconditions would probably require interaction with the instructor in order to identify the conditions that differentiate the disjuncts and to associate each positive example with the appropriate disjunct.

8.4.3.2 More Involved Extensions

The above extensions are reasonably simple. In this section, we will talk about extensions that would involve larger changes to Diligent.

Use structural knowledge. Diligent may have an unstructured environment. An *unstructured environment* contains a set of attribute values without any indication of the relations between attributes and objects. While Diligent's techniques can work in a structured environment, the techniques do not take advantage of knowledge about the environment's structure. If Diligent used knowledge of how attributes were associated with objects and how objects were related, it might be able to do a better job of learning preconditions. This knowledge would allow Diligent to focus on the attributes of objects being manipulated by actions; this might be useful because some of these attributes are likely to be important. But more importantly, structural knowledge would make it easier to generalize operators so that they could apply to a class of objects, contain variables, or even use relations between objects.

Use a deeper domain model. When Diligent starts working on a new domain, it has no knowledge of the domain. It would be interesting to see how Diligent's approach to understanding demonstrations could be modified to exploit access to a deeper domain model.

8.4.4 Experimentation

In the chapter on experimentation, experiments were loosely defined as activities initiated by the system that acquire more knowledge. These activities included autonomously manipulating environment as well as querying the user for more information.

The extensions to Diligent's techniques fall into two categories. One group contains extensions that follow naturally from Diligent's approach. The other group contains more involved extensions that could complement Diligent's approach.

8.4.4.1 Simple Extensions

The following extensions follow naturally from Diligent's approach.

- If the system has not seen enough examples of an action producing a desired state change, then ask the instructor for more data. This extension is inspired by Galdes [Gal90] study of expert human tutors.

- Notice when a subprocedure's internal step relationships change. This can happen when the instructor explicitly works on the subprocedure, but it can also happen during experiments or during demonstrations when a subprocedure is inserted into another procedure. These situations are a good source of experiments and a good place to interact with an instructor.

- Notice when a subprocedure unexpectedly misses its goal conditions. At this point, Diligent needs information about where to add steps or why unused steps are necessary. This extension is also inspired by Galdes [Gal90] study of expert human tutors.

- When all else fails, interact with the instructor. For example, if some necessary preconditions are missing, the system may continue to classify a negative example as positive. In this case, the instructor could be presented with several attribute values and asked about their importance.

  Interacting with the instructor to classify examples is explored in much greater depth by MOLE [EEMT87], which learns diagnostic knowledge from an expert by focusing on how to classify situations and differentiate between hypotheses.

  Other work has looked at engaging the instructor in a dialog in order to determine which action to perform in a given situation [HL95, Gru89].

8.4.4.2 More Involved Extensions

The following extensions are more involved than the ones in the previous section, but could complement Diligent's approach in some future system.

- Diligent avoids asking the instructor questions. However, the instructor's assistance could help when the system is confused and the number of questions and answers is limited. The PRODEGE+ graphics editor [BS93] explores this type of dialog.

- After Diligent has experimented on demonstrations, the system has better knowledge of operators. At this point, it may be appropriate for the system to experiment by creating plans. This could involve explicit experiments where the environment is put in a specific state so that action can be tested, or it could involve solving practice problems where the environment is transformed from some initial state into a specified goal state.[7]

  With human students, a similar approach is often used. They examine the solutions of few problems before solving a some related problems.

8.5 Summary

We started the chapter by discussing Diligent's methods in terms of different perspectives for viewing demonstrations. We then discussed assumptions and how easily they could be relaxed. We finished by discussing various limitations and potential extensions.

[7]This use of plans was discussed in Section 6.2.3.

Chapter 9

Related Work

Throughout this document we've discussed related work where appropriate. This chapter covers other work that hasn't been discussed. The chapter focuses on three somewhat separate topics. The first topic is how to present examples in order to promote learning. The second topic is intelligent tutoring systems. The third topic is systems that learn from demonstrations. (Although many systems that learn from demonstrations have already been discussed, they have not been discussed as a group or as complete systems.)

9.1 The Presentation of Examples

Because demonstrations are the primary input that Diligent receives from instructors, we will briefly look at other work that deals with the presentation of similar types of data. We will first discuss properties of good instruction and then discuss how to present examples.

9.1.1 Felicity Conditions

Good instruction of human students follows a set of conventions. VanLehn [Van83] characterizes some of these conventions and calls them *felicity conditions*. VanLehn uses the felicity conditions in SIERRA [Van83, Van87], a system that models human students learning subtraction.

One difference between SIERRA and Diligent is the nature of their inputs. SIERRA receives an ordered sequence of lessons where each lesson can contain solutions to multiple similar problems. In contrast, Diligent receives a sequence of demonstrations, and each demonstration corresponds to a lesson that contains the solution to only one problem.[1]

[1]The types of demonstration's supported by Diligent are described in Section 4.2.

In the following discussion, keep in mind some of the differences between Diligent and human students. Humans require reinforcement and repetition of what they have learned, while Diligent never forgets. One advantage Diligent has is its access to a simulation, which can be used to perform experiments. Usually, human students don't have access to the equivalent of Diligent's simulation (e.g. when they are learning subtraction). Determining the beliefs of human students is much more difficult for a teacher than it is for Diligent's instructor, who can use menus to look directly at Diligent's knowledge.

A natural question is how well does the relationship between Diligent and the instructor match VanLehn's felicity conditions. Let us consider each of the felicity conditions.

- *Assimilation.* A procedure is incrementally improved by adding to the existing procedure without revising large portions of it. VanLehn writes, "incremental learning is an important and nearly universal feature of human skill acquisition"([Van83], page 10).

  Diligent's add-step demonstrations guarantee this felicity condition because the instructor indicates where to insert steps in an existing procedure. However, demonstrations can have a large impact when they alter step relationships.

  Diligent's clarification demonstrations do not have an equivalent in SIERRA. A clarification demonstration provides data for machine learning without adding steps to a procedure's plan. However, clarification demonstrations also incrementally improve a procedure by refining operator preconditions.

  Because Diligent never forgets and does not get confused when switching between contexts, Diligent does not have the problems that humans do when large portions of a procedure are changed. This means that a machine learning system, like Diligent, may not need to follow this felicity condition. (However, following it may seem natural to an instructor.)

- *Generalization.* During a lesson, one way that a student learns is by generalizing the lesson's example solutions.

  Diligent also does this when it uses machine learning techniques to learn preconditions. However, unlike a human student, Diligent does not make generalizations for whole classes of objects (e.g. how to log into all computers).

- *Show work.* At least in introductory lessons, all work should be shown.[2]

[2]SIERRA also has lessons that optimize an existing procedure by showing how to eliminate unnecessary work, but this type of lesson may not be necessary.

Diligent's instruction meets this condition. Diligent sees all relevant attributes of the environment and observes all actions performed in the environment. In fact, it is easier for Diligent's instructor to meet this felicity condition than it is for someone who teaches human students.

- *One disjunct per lesson.* In each lesson, the student should need to add at most one disjunction to his mental model of a procedure. A disjunct contains a sequence of actions and a conditional test to decide whether to perform the actions. VanLehn sometimes refers to disjuncts as subprocedures.

  Because of different procedural representations, this is harder to characterize. One of Diligent's add-step demonstrations could be considered a disjunct because an add-step demonstration contains a sequence of steps that are inserted between existing steps. However, Diligent learns preconditions for each step rather than one for the entire demonstration.

  This felicity condition does not apply to clarification demonstrations because they don't add steps to the procedure's plan and because they can contain arbitrary sequences of steps. Unlike a human, Diligent can use clarification demonstrations because it does not forget and does not get confused when switching between contexts. Clarification demonstrations can be thought of as exploratory demonstrations in which an instructor illustrates the behavior of the environment.

Other people have adapted VanLehn's felicity conditions. When discussing felicity conditions, Wenger [Wen87] includes the condition *minimal set of examples.* This means that example solutions are sufficient to learn the new subprocedure.

However, Diligent does not assume that the instructor provides a minimal set of examples. Instead, Diligent uses heuristics to create a "reasonable" procedure that the instructor can then examine, edit and test. In this sense, Diligent, without instructor input or critique, is not expected to achieve the mastery or proficiency of human students, which makes Diligent's task much easier.

The felicity conditions have also been adapted for Programming By Demonstration (PBD) systems [CKM93]. Because Diligent uses PBD, these conditions are relevant.

- *Be consistent.* The steps in a demonstration need to be performed consistently in the same order.

Consistency is important for typical PBD systems because they only need to know how to automate a procedure. Because these systems do not usually have access to a simulation, they use induction to learn how to sequence a procedure's steps.

In contrast, Diligent attempts to acquire the knowledge necessary for teaching, which requires more knowledge about the dependencies between steps. Diligent not only needs to be able to answer questions, but it must also be able to monitor students as they perform a procedure. Because students may legitimately perform steps in a different order than any demonstration, Diligent needs to be able to recognize whether an alternative sequence of steps will achieve a procedure's goals.

Diligent can violate this felicity condition because it uses a simulation to induce operator preconditions that are independent of the current procedure. Later, when creating a plan, Diligent uses these preconditions to analytically derive the dependencies between steps. Because operator preconditions are not procedure specific, Diligent's clarification demonstrations do not cause a problem when they violate the "be consistent" felicity condition. In fact, clarification demonstrations are meant to violate this felicity condition.

- *Correctness.* The procedure is correctly demonstrated.

  Diligent makes this assumption. If this assumption is violated, then Diligent can still learn, but the preconditions of its operators may not be as good.

- *No extraneous activity.* An extraneous step might not be incorrect, but it doesn't contribute to the goal. One problem is that extraneous activities are likely to confuse or mislead a typical PBD system.

  Diligent can compensate for extraneous activity because it has access to a simulation. While extraneous steps in add-step demonstrations are undesirable, Diligent learns to skip them through its experiments. Thus, extraneous steps are not usually a problem for Diligent. Furthermore, this issue is not relevant for clarification demonstrations because they do not add steps to plans. In fact, extraneous activities are probably beneficial in clarification demonstrations because they provide more data for learning.

9.1.2 Presenting a Sequence of Examples

Other work by Mittal has focused on how to present sequences of examples to humans [Mit93, MP93]. This raises two issues: how well do the inputs given Diligent's instruction meet these criteria and how do Diligent's abilities compare to a human student's.

In Mittal's work, "example" is used to describe the training data, and both Diligent's action-examples and demonstrations (i.e. sequences of action-examples) would be considered "examples."

Mittal [Mit93] looked at many issues involved in the presentation of examples. Of these issues, the following appear to be relevant.

- *Minimum detail.* Studies have shown that people learn best when examples contain a minimum number of irrelevant features. Mittal calls this the minimum detail principle.

  Diligent's action-examples do not meet this requirement; Diligent learns in an environment with a constant number of attributes. However, Diligent uses the minimum detail principle when it heuristically focuses on a small number of likely operator preconditions. For example, when creating an operator, Diligent assumes that the state changes of the demonstration's earlier steps are good candidate preconditions for the new operator.

  The principle of minimum detail applies to Diligent's add-step demonstrations, which should not contain unnecessary steps. The principle also applies to Diligent's environment during demonstrations because only the instructor is performing actions.

  Like a human, Diligent learns best when there are few irrelevant details, but unlike a human, Diligent not forget and can save negative action-examples until it is able to process them.

- *Number of examples.* If humans are given too many examples, they tend to have lapses of attention.

  Lapses of attention are not an issue with automated systems, and Diligent learns best with many examples.

- *Order of presentation.* The order of presentation helps avoid confusion and focuses the student's attention. Simpler, more easily understood examples should be presented before more complicated and difficult examples. This is closely related to VanLehn's assimilation and one disjunct per lesson felicity conditions (Section 9.1.1).

  Diligent learns best when its demonstrations represent small, modular and logically coherent procedures. However, since Diligent uses state changes from within a demonstration to identify likely preconditions, Diligent should learn better when it has a long logically coherent demonstration rather than several incrementally more complicated procedures.

- *Pairing of examples.* An example should highlight some feature. This means that there is a relationship between an example and the principle being taught.

  Mittal identifies three types of examples: A *positive* example is instance of the concept being taught; a *negative* example is not an instance of the concept; and an *anomalous* example represents a special case or an exception.

  Diligent processes positive and negative examples, but makes no special provision for anomalous examples. Anomalous examples are treated like any other example. If Diligent were to make special provision for anomalous examples, it would probably have to support disjunctive preconditions. For example, a device might normally be reset by pressing the reset button, but while in test mode, it might only be reset by pressing the system test button. In this example, resetting the device in test mode is an anomalous or special case.

  Mittal discusses how pairing different types of examples teaches different principles. Pairing two positive examples allows students to identify unnecessary (or *variable*) features. Pairing a positive and a negative example allows students to identify necessary (or *critical*) features. Furthermore, pairs of positive examples should be as dissimilar as possible, while a positive and negative example should be as similar as possible. In fact, Mittal writes that studies [Fel72, HMD73, KGF74, MT69] suggest that the most effective pairing of examples are minimally different positive and negative examples.

  Like human students, Diligent's algorithms for refining operator preconditions also do best with maximally different positive examples and minimally different positive and negative examples.

  However, Diligent's demonstrations do not necessarily provide this type of data. An add-step demonstration only provides one action-example for each step. For a clarification demonstration, it is entirely dependent on the instructor whether the demonstration provides dissimilar positive examples or similar pairs of positive and negative examples.

  Diligent overcomes its lack of action-examples by performing experiments. Experiments are derived from demonstrations and tend to produce similar pairs of positive and negative examples.

  However, without the help of the instructor, Diligent cannot create very dissimilar pairs of positive examples. One obstacle is the minimal assumptions that Diligent makes about its ability to manipulate the environment. This problem might be addressed by using planning techniques to create more elaborate experiments.

Mittal also addresses a couple of issues that don't map well to Diligent. One issue is how advanced is the material. Because Diligent receives action-examples with a fixed number of attributes rather than increasing numbers of attributes, Diligent's input doesn't correspond well the increasingly detailed training given humans. Another type issue is the type of knowledge being taught. While Diligent can learn about relationships between inputs and outputs (i.e. operators) and sequences of relationships (i.e. procedures), Diligent does not learn the types of concepts that a human learns (e.g. apples grow on trees).

9.2 Intelligent Tutoring Systems

Because Diligent creates procedures for a tutoring system, we need to discuss tutoring systems and authoring systems for tutoring systems. We will first discuss computer systems that provide instruction, and we will then discuss issues and approaches for authoring.

9.2.1 Computer Aided Instruction

Traditional forms of *Computer Aided Instruction (CAI)* require authors to create a fully specified presentation of the material, including questions and answers [Wen87, Ric89, Mur97]. This includes specifying the flow of control through the material. Because the material is grouped into fixed blocks or "frames" of knowledge, traditional CAI has been referred to "electronic page turning" [Ric89].

While CAI is useful for some types of instruction, it has problems. CAI systems tend to be inflexible and allow only limited tailoring of instruction to individual students. The problem is that CAI systems know little or nothing about what is contained in the frames.

A reaction to traditional CAI is *Intelligent Tutoring Systems (ITS)* [Wen87, Ric89]. A main distinction between CAI and ITS is that, instead of using CAI frames, ITSs use the knowledge that was used to compose the frames [Wen87]. A primary characteristic of ITS is using this knowledge for multiple purposes. For example, the same piece of knowledge might be used for presenting material, formulating a question and answering it. Consider the STEVE tutor, which is used with Diligent. STEVE uses plans to demonstrate procedures, monitor students as they perform procedures, answer student questions, and recover from student errors. STEVE couldn't do this if it just knew about a fixed sequence of steps.

ITS research has focused in a number of areas. One area is modeling the student's knowledge. The model may include what students have seen as well as what the system believes about their knowledge [SS98, Wen87, Sel74, Car70]. Another area is modeling different teaching strategies; this includes how to present material, what type of questions to ask, and when to intervene [MAW97, Maj95, Hil94, SJ91, Wen87]. And a third area is using simulations to provide students with a richer, more complex and interactive learning environment [MJP$^+$97, VD96, Wen87].

In this thesis, we have focused on authoring procedures for use with a simulation. We have ignored student modeling and teaching strategies because we have assumed that an automated tutor would already have knowledge of these activities.

Another problem with CAI systems is that authoring these systems takes a long time. According to Woolf and Cunningham, each hour of instruction typically requires 200 hours of development [WC87].

Ideally, by reusing knowledge, authoring knowledge for ITSs should be simpler than for a CAI, but this is not so. Not only do ITSs have an additional capabilities, which require additional knowledge, but their knowledge also needs to be more structured. In fact, Murray [Mur97] has written that one of the biggest problems with ITS research is that ITSs are "difficult and expensive to build." For this reason, ITS authoring is an active area of research.

The next few sections will discuss ITS authoring issues.

9.2.2 Who is the Author

A primary concern when considering an ITS authoring tool is what type of person will do the authoring. Is a tool designed primarily for experienced, expert users or is it designed for wider class of user? This is important because different tools are designed for different types of users. When considering various approaches to ITS authoring, we will consider two types of authors.

- An instructional designer provides materials for many teachers and students. An instructional designer may have specialized training in instructional design and in the use of authoring tools. However, an instructional designer might have little interaction with teachers or students.

- A teacher authors material for his class. The teacher is unlikely to have the same specialized training as an instructional designer and will most likely have limited time for authoring. However, unlike an instructional designer, a teacher should have a lot of interaction with students.

Diligent focuses on quick and easy authoring of procedures so that its techniques could be used by a large class of users that includes both instructional designers and teachers.

9.2.3 Approach to Authoring

Because of the difficulty in creating an ITS, researchers have tried different approaches. Below are some of the basic ITS authoring approaches.

- *Monolithic/evolutionary.* These systems contain everything needed for instruction. This type of system attempts to incrementally evolve the state of the art of commercial CAI authoring tools. The system is usually targeted towards instructional designers. For example, EON adds improved modularity and abstraction to a CAI approach [Mur98]. In contrast, the IRIS Shell [AFCFG97] structures authoring around Gagne's theory of instructional design [GBW88].

  A problem with this type approach is the time involved. For example, Murray [Mur98] reports the success of an earlier ITS authoring tool that supported authoring an hour of instruction in 100 hours. He compares this favorably to the 100 to 300 hours of a traditional CAI approach.

  However, using this type of system doesn't have to be laborious. REDEEM [MAW97, MA97] is targeted towards teachers rather than instructional designers. REDEEM allows teachers to reuse the content of an existing CAI course and to tailor the teaching strategies used with individual students. When given the CAI data, REDEEM appears easy to use.

- *Framework.* This type of system asks the instructor to provide data for use in a predefined instructional framework. The author will provide predefined types of data, and the system will reuse predefined pedagogical knowledge. This type of system is also monolithic.

  Much of the work in this area has been done at Northwestern and has focused on Goal Based Scenarios (GBS) [Sch94, JK97, Bel98, DR98]. GBS systems have students work on several scenarios using the method determined by the given framework. For example, the Investigate and Decide framework requires students to make a decision after investigating the situation with a set of tools. Another framework, Persuade, lets students interact with simulated characters and to build a consensus or to change the positions of the simulated characters.

To author a GBS, the author provides scenarios, tools, video clips, questions and answers. The GBS framework will then integrate the data when providing instruction. Unfortunately, authoring with these systems can take several weeks [Bel98] or from 5 to 10 months [DR98]. Because of time involved, a teacher is unlikely to author with existing GBS frameworks.

In contrast, XAIDA allows quick authoring (i.e. minutes to hours) [Red97, HHR99]. Little work is needed because XAIDA has a great deal of knowledge about how to present machine maintenance training. XAIDA focuses on what to present rather than how the domain works. For example, to teach a device's physical characteristics, the author labels portions of a picture and provides simple knowledge (e.g. the function of a part). However, XAIDA is self-contained and cannot interact with a complex simulation of the device.

- *Component.* In this framework, a heterogeneous group of tools interact [RK96, RB98, JRSM98]. Not only can high quality components be developed independently, but components can potentially be reused on other systems. However, when using this approach, knowledge is localized inside the components. For example, one component may know a great deal about teaching but little about the domain.

  This dissertation deals with the component framework and focuses on helping an author exploit the domain knowledge already contained in other components.

9.2.4 Easier Data Entry

All ITS authoring research focuses on making ITSs easier to author, but most work has focused on supporting the additional capabilities not found in CAI. Relatively few systems have focused on data acquisition with machine learning techniques or using extremely quick authoring. Systems that acquire knowledge quickly can do so because they focus on acquiring shallow knowledge about well-defined and constrained activities.

One system that we've discussed is XAIDA [HHR99], which knows a great deal about generating instruction. XAIDA uses data provided by the instructor to instantiate a generic instruction template.

Another system, DIAG [Tow97b, Tow97a], focuses on teaching fault diagnosis. DIAG generates a probabilistic table of faults by modifying a simulation. Unlike Diligent, DIAG is contained in the simulation and can directly access and modify it.[3]

Demonstr8 [Ble97] can author an ACT tutor [A^{+}95]. Demonstr8 allows the author to create the student's interface using the Graphical User Interface (GUI). Demonstr8 also induces expert behavior from examples. However, the version of Demonstr8 described in the paper can only create simple arithmetic tutors. It is unclear how easily the system can be scaled to domains where functions are not simply lookup tables.

Recently, Disciple [TH96, TK90] has been used to inductively learn how to classify examples of a given concept [TK98]. Disciple first has the author build a semantic net of object classes and relationships. Then, the author provides Disciple with examples of a concept. Finally, Disciple asks the author whether other examples are members of the concept's class. Although Diligent learns procedures rather than individual concepts, the preconditions of Diligent's operators are similar to Disciple's concepts. Unlike Disciple, Diligent does not use a semantic net and can perform experiments that query the environment rather than the author.

Work at the University of Pittsburgh's Learning Research and Development Center has looked at using human style reasoning to learn how to solve procedural problems (e.g. physics problems) [GCV98].[4] This approach requires access to well-defined domain rules (e.g. physics laws) and problem modeling techniques. In contrast, Diligent is made for domains where this type of knowledge is not readily available.

Although not strictly an ITS authoring tool, ODYSSEUS [Wil90, Cla86, Wen87] learns knowledge about medical diagnosis that can be used by the GUIDON family of ITSs. ODYSSEUS learns best by observing a physician make a diagnosis. It then attempts to explain the diagnosis using a domain model and a diagnostic strategy model. If an explanation is not found, it uses heuristics to make inductive changes to the domain model. ODYSSEUS is able to update the domain model because it uses a known problem solving strategy and assumes that the domain model is almost correct. The validity of the approach was demonstrated in an experiment [Wil90]. After observing only two diagnoses, ODYSSEUS showed a 37% improvement in its ability to make a correct diagnosis. This

[3]DIAG is implemented in RIDES, which is an ITS authoring tool for simulations. After a simulation has been built, RIDES also supports quick authoring of instruction. RIDES is discussed in section 9.3.1.

[4]The work at the University of Pittsburgh has explored the use of the human Self-Explanation Effect, which was discussed in Section 6.8.1.

improvement occurred even though the physician misdiagnosed one of the two cases. However, ODYSSEUS differs from other systems in this section because it uses a deep domain model. For example, the model used in the experiment was acquired over seven year period [Wil90].

9.3 Learning From Demonstrations

This section talks about systems that learn from demonstrations. Specifically, it discusses systems that learn from traces. A *trace* is a record of the procedure being performed. If this type of system learns by observing users carry out their normal activities, the system is called a *Learning Apprentice System* (LAS) [MMS90].

9.3.1 Programming By Demonstration

Diligent's use of demonstrations to learn procedures is called Programming By Demonstration (PBD) [C+93]. Unlike simply recording a macro, PBD by definition requires some generalization. Diligent is an unusual PBD system in that it generates data by performing autonomous experiments.

PBD has been used for several types of purposes, such as creating user interfaces and learning procedures. We will focus on PBD systems that learn procedures.

A traditional PBD system learns procedures in order to automate tasks. This involves making procedures work on multiple objects and determining which conditions indicate a change in a procedure's flow of control. A condition that indicates a change in the flow of control is called a *branch condition*. An example of a branch condition is a condition that indicates whether to exit a loop. However, traditional PBD systems do not attempt to learn in detail how steps depend on each other (i.e. step relationships). This means that they would not be able to recognize whether a different sequence of steps was valid or to provide explanations about the dependencies between steps.

The actions in Diligent's demonstrations bear a lot of similarity to those of robotic PBD systems [FMD+96, Hei93, Hei89, And85]. However, these systems concentrate on eliminating sensor noise and finding loops and branch conditions. Like traditional PBD systems, these systems learn to perform a task without learning the step relationships required for the type of teaching that Diligent supports.

A system that can use demonstrations to learn similar types of procedures as Diligent is the RIDES [MJP$^+$97, MJSW93] authoring system.[5] RIDES is one of the most used ITS authoring systems. It supports authoring of graphical simulations without a great deal of programming expertise. RIDES also supports the ability to enter many types of training exercises, and it is the training exercises that are relevant to Diligent.[6] While training exercises use the executable simulation model, training exercises are separate objects that contain little knowledge about the model. Unlike Diligent's plans, these training exercises do not contain detailed knowledge about the dependencies between steps (i.e. causal links). Because less has to be known about the procedure, it is much easier to demonstrate in RIDES than it is in Diligent. Authoring with RIDES involves demonstrating the procedure and interacting a little with menus. However, because RIDES' exercises lack causal links, RIDES can only provide limited help and remediation.

9.3.2 Detailed Domain Models

An early robotic demonstration system that only requires one demonstration is ARMS [Seg87]. Unlike Diligent, ARMS relies on a detailed domain model and a geometric reasoner to deduce a procedure's structure.

Like ARMS, another system that uses a detailed domain model is LEAP [MMS90]. LEAP uses its theoretical knowledge of circuits for learning how to implement components of a circuit. LEAP relies on *Explanation Based Learning* (EBL) [MKKC86, DM86] and can only learn when its domain theory can explain a training example. In contrast, Diligent starts with little domain knowledge and focuses on acquiring the domain theory necessary for explaining a procedure.

LEX [MUB83] does not learn procedures; instead, it is given a set of operators and learns when to perform them. LEX starts knowing a set of mathematical transforms that it uses to solve symbolic integration problems. These transforms are analogous to the operators that Diligent learns. Instead of receiving traces as input, LEX uses the solutions to problems that it has solved. LEX is relevant because it tries to maximize the use of its limited problem solutions by minimally modifying the problem and then attempting to solve it.

[5] Diligent's environment is controlled by a version of RIDES called VIVIDS.

[6] RIDES' procedure and goal patterned exercises are similar to Diligent's procedures.

CELIA [Red92] can learn machine maintenance procedures similar those learned with Diligent. However, instead of learning procedures for teaching humans, CELIA models human performance and learning. Because CELIA contains a detailed but possibly incomplete domain model, CELIA, unlike Diligent, is able to learn complicated troubleshooting tasks. CELIA receives high level English descriptions of diagnostic procedures and can ask the user questions when it gets confused or discovers problems in its domain model. Because CELIA emphasizes reducing gaps in its knowledge, CELIA only learns when a failure identifies missing knowledge. In contrast, Diligent can learn from both success and failure because it attempts to reduce the uncertainty in its knowledge. CELIA focuses on learning how the diagnostic goals of a procedure's steps are related rather than the low-level preconditions that Diligent uses for creating step relationships. Like many case-based systems, CELIA's indexing of the steps of a procedure is very dependent on the order that CELIA receives training examples.

9.3.3 Procedure Recognition

Two systems that require traces of slightly increasing complexity are SIERRA [Van87, Van83] and NODDY [And85]. SIERRA models children learning subtraction and creates procedures in the form of AND-OR graphs, while NODDY, an early PBD system for two dimensional robots, learns procedures in the form of flow charts. Both systems learn incrementally but non-interactively from traces. These systems learn by matching their model of a procedure against a trace to find differences. Because these systems need to match existing procedures, they rely on the user adding little complexity per trace. For example, in Section 9.1.1 we discussed SIERRA's assumption that the instructor adds only one disjunct per lesson. Diligent avoids this problem by requiring the instructor to specify the position where steps are inserted. Because SIERRA and NODDY learn the control knowledge necessary to a perform procedure rather than operators that model the domain, they can only use traces that illustrate how to perform a procedure and could not use traces similar to Diligent's clarification demonstrations. Additionally, neither system refines its knowledge with experiments.

9.3.4 University of Michigan Soar Group

Soar [LNR87] is a production system that implements a unified theory of human cognition [New90]. Diligent was written in the environment of the Soar community. In fact, the tutor used with Diligent, STEVE, is implemented primarily as Soar productions. The work on instructable agents in Soar at Michigan heavily influenced Diligent's interaction with the instructor.

Instructo-Soar [HL95, Huf94, HL93] receives tutorial instruction in a manner similar to Diligent but in English rather than by direct manipulation. Unlike Diligent, a user can tell Instructo-Soar what to do in hypothetical situations (e.g. "when the light is red, press the green button"). Unlike Diligent, which learns operators, Instructo-Soar is given set of general-purpose operators that model actions performed its domain. Unlike Diligent, Instructo-Soar does not modify its operators and does not refine its knowledge by performing autonomous experiments. Instead of learning plans, Instructo-Soar uses its operators to learn when to reactively perform actions (i.e. operator proposal rules). If Instructo-Soar's operators were correct, it could generate the step relationships that Diligent learns. However, if Instructo-Soar's operators were incomplete or incorrect, then it would have problems generating Diligent's step relationships. If Instructo-Soar's operators cannot explain a demonstration, it uses heuristics to create operator proposal rules that allow it to correctly perform a procedure.

Instructo-Soar has been extended by IMPROV [PL96, Pea96], which refines its knowledge with experiments. IMPROV performs procedures and refines its knowledge when failure is detected. Unlike Diligent, IMPROV can handle noise and work in dynamic domains whose properties change. IMPROV experiments by performing actions in the environment during a search for a sequence of steps that achieves a procedure's goals. In contrast, Diligent can learn without failure and doesn't care if its experiments achieve the procedure's goals. When IMPROV finds a successful plan, it learns to perform the plan's steps in the same order as the successful plan. IMPROV does this by learning reactive rules to propose operators. The problem with this approach is that it doesn't learn about alternative orders of steps that would also achieve the goals. This means that IMPROV's approach doesn't learn good preconditions for deriving Diligent's step relationships. Another problem is how IMPROV represents its reactive rules. IMPROV never forgets a rule even though it may have missing or unnecessary preconditions; instead, IMPROV creates a patchwork of overlapping, prioritized rules. It appears likely that a human instructor would find this representation difficult to comprehend and verify.

9.3.5 Approach to Experimentation

Diligent's approach to experimentation is most similar to PET's approach [PK86]. Unlike Diligent, PET learns *relational rules*, which use arbitrarily complex domain dependent transformations to change the state before the action into the state after the action. In contrast to Diligent, which modifies a demonstration by changing the actions that are performed, PET modifies a demonstration by changing the state. PET's approach to experimenting requires complete control of the state and involves repeatedly performing an action after making fine grain changes to the state. Because Diligent does not have complete control over the state, it could not use PET's approach.

9.3.6 Systems that Learn Operators

Operators model actions performed in the environment and identify the preconditions necessary to produce various state changes. Diligent is unusual in that it learns operators that are only applied to a few instructor specified procedures. In contrast, other systems learn operators for solving general planning problems. These systems experiment by solving practice planning problems, where an initial state is transformed into a goal state. In contrast, Diligent experiments by modifying its demonstrations. Diligent doesn't care about an experiment's final state because its experiments focus on identifying dependencies between the given procedure's steps.

A system that systematically refines its operators is EXPO [Gil92, CG90]. EXPO refines operator preconditions when an unexpected state change is observed while solving planning problems. Unlike Diligent, EXPO is given a set of incomplete operators with their preconditions partially specified. EXPO then refines its operators by adding preconditions. Unlike Diligent, EXPO cannot remove incorrect preconditions. EXPO introduces general heuristics for proposing preconditions that rely on the similarity of objects and the relationship between objects and actions. Unlike Diligent, EXPO can also learn a new procedure by an analogy to an existing procedure that uses similar classes of objects. In contrast, Diligent does not have a hierarchy of object classes, and many of Diligent's objects (e.g. switches) have idiosyncratic behavior that prevents reuse of operators with different objects (e.g. switch1 turns on the motor, while switch2 turns on a light).

A system that heavily influenced Diligent is OBSERVER [Wan96c, Wan96a, Wan95, Wan96b]. Unlike Diligent, OBSERVER generalizes the objects and attributes in its operators. Diligent doesn't do this because it has less knowledge of its environment and many objects in its environment have idiosyncratic behavior. OBSERVER learns operators by observing traces of many demonstrations and solving many planning problems. In contrast, Diligent has only a few demonstrations and does not solve planning problems. Unlike Diligent, OBSERVER does not consider the relationship between steps in a demonstration when hypothesizing preconditions.

9.3.7 Other Work

A system that learns a different type of operator than Diligent is TRAIL [Ben95]. TRAIL processes demonstrations and uses inductive logic techniques to learn reactive teleo-operator proposal rules. *Teleo-operators* [BN95] model actions that can have a duration. Unfortunately, TRAIL learns only one definite state change per operator. The operator's other state changes have a probability of appearing. This would be unacceptable for teaching procedures in a domain where an action can change the values of multiple attributes. It might be possible to learn different conditional effects for different state changes; however, because a conditional effect's state change is definite, it is unclear whether TRAIL's probabilistic learning algorithm would still be useful.

Recent work by Bauer [Bau98] takes a different approach for understanding traces. Unlike Diligent, which focuses on the attributes in the environment, Bauer looks at acquiring plans using relationships between arguments of different actions. For a number of reasons, this approach is inappropriate for Diligent. The program that is learning procedures (e.g. Diligent) may not know how some objects are related to each other. This means that performing an action may cause changes in distant objects that appear to be unrelated to the object acted upon. For example, pressing a button may turn on a fan in another room. Furthermore, in Diligent's procedures, every step may manipulate a different object; thus, the arguments to a procedure's actions often have little commonality. Finally, Diligent produces procedures for a type of tutoring that requires causal links between steps, and causal links are derived from the preconditions of steps. Some of these preconditions may involve the environment's state rather than the properties of the objects being manipulated.

LIVE [She93, She94] is a system that uses autonomous exploration and experiments by creating plans. Because LIVE doesn't focus on learning user specified procedures, it is unclear how well it would scale to more complex domains because of the time involved and the lack of focus. Because LIVE doesn't process traces, its main relevance is its machine learning techniques. Besides experimenting with plans, LIVE learns rules to predict when an action will produce given state changes; these prediction rules are similar to the preconditions of Diligent's conditional effects (or effects). Unlike Diligent, LIVE requires structural domain knowledge and only learns from prediction failure. LIVE's approach for learning prediction rules, Complementary Discrimination Learning (CDL), updates prediction rules by comparing the prediction rules for different effects. However, the updated rules can contain both disjuncts and negated conditions. When compared to Diligent's simple conjunctive preconditions, the representation of prediction rules may seem overly complex to a human instructor.

Chapter 10

Conclusion

In this last chapter, we will summarize this thesis and its contributions. We will also discuss some potential future work.

10.1 Summary of the Approach

This thesis looks at the problem of authoring procedures for an automated tutor that is used in a heterogeneous, simulation-based training environment. To teach, the automated tutor needs certain capabilities. It must be able to demonstrate procedures for human students, monitor students as they perform procedures, answer questions about a procedure, and recover from student errors and unusual environment states. Monitoring students is difficult because students may use a valid sequence of steps that is different than what was demonstrated, and answering questions is difficult because missing or incorrect information causes confusion. It is assumed that the tutor has general knowledge of how to teach, but is missing knowledge of the procedures that it teaches.

Unfortunately, acquiring knowledge from domain experts (e.g. instructors) can be difficult. Domain experts may not be programmers or expert knowledge engineers. Therefore, Diligent exploits the presence of a simulation to make authoring easier. The techniques explored in this thesis could potentially allow non-programmers to author procedures by demonstrating them with a graphical interface that represents the state of a simulation.

Less work is required from an instructor because Diligent uses the simulation to perform experiments. These experiments allow Diligent to get answers to questions from the simulation instead of the instructor. Because Diligent can answer its own questions, not only is there less chance of instructor error, but Diligent also needs fewer demonstrations. Because less data is required from the instructor, the difficulty of authoring is also reduced.

One way that Diligent's techniques could help an instructor is by providing feedback about its beliefs. For example, Diligent uses three sets of preconditions (i.e. s-rep, h-rep and g-rep) and each set represents a different level of confidence. When users look at preconditions, Diligent indicates its level of confidence that a given precondition is necessary. For example, preconditions that only appear likely (in h-rep but not in g-rep) have lower level of confidence than preconditions that have been shown to be necessary (in g-rep).

Because Diligent may have very little knowledge, it uses heuristics to speed up learning. It assumes that the state changes of earlier steps are likely to be preconditions of later steps. It also uses an heuristic, best guess precondition concept (i.e. h-rep) that is in between the the upper and lower bounds of its version space. Unlike the version space bounds, the h-rep supports error recovery by allowing preconditions to be both added and removed.

Diligent also bounds the cost of experimentation. Its experiments change the order of a procedure's steps by skipping a step and observing what happens to later steps. Because the purpose of an experiment is to perform the steps rather than to achieve some goal state, experiments perform a bounded of number of steps. Additionally, in experiments on hierarchical procedures, Diligent only experiments on the current procedure and treats subprocedures of the current procedure as single steps.

A nice aspect of Diligent's approaches to experimentation and to learning operators is that they balance each other. When operators are created, the preconditions tend to have errors of commission (i.e. unnecessary preconditions). On the other hand, by skipping steps, experiments tend to identify errors of commission. Furthermore, in Diligent's version space learning algorithm, errors of commission are easier to eliminate than errors of omission (i.e. missing preconditions).

10.2 Contributions

The main contribution are the following.

- 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.

A lesser highlight of the thesis is that it also presents algorithms that show how to transform demonstrations into hierarchical partially ordered plans. These algorithms, additionally, provide the framework that supports learning operators and performing experiments.

10.3 Evaluation

An empirical evaluation using human subjects was performed (Chapter 7). The evaluation looked at the benefits of both demonstrations and experiments. The analysis of the study focused on contrasting a simple versus a complex procedure. The study suggested that both experiments and demonstrations help, and they help more on complex procedures.

10.4 Future Work

Earlier in Chapter 8, we discussed a number of extensions. Some of the extensions for demonstrations required multiple paths (or sequences of steps) for performing a procedure. This would allow additional types of demonstrations and more complicated procedural representations, including conditional plans. Some of the extensions for machine learning include supporting disjunctive preconditions, using structural knowledge and using a deeper domain model. Some of the extensions for experiments include practice problems and modifying experiments in response to unexpected events.

However, the techniques discussed here could be used for other purposes.

Diligent's techniques could help systems that learn general-purpose operators for planning by helping them to better understand demonstrations and the solutions to practice problems, which could be treated like demonstrations.

In Diligent's current project, procedures are learned for a tutor that uses a virtual environment. However, Diligent only requires a graphical interface and not a three dimensional virtual environment. One potential application is creating procedures that teach people how to run a factory using a two dimensional display of various controls and indicators.

Diligent could also be used by students who are attempting understand a device. Students could identify the state changes produced by manipulating various controls. Students could also use Diligent to learn preconditions and to learn procedures.

Another use is debugging simulations, especially when the simulation is developed by external organization. A major problem with simulations is that it is often difficult to determine what type of calculations they perform internally. This means that it is difficult to know how normal results are reached or what the simulation will do in unusual situations. A non-programmer, domain expert could test a simulation by authoring procedures and looking at looking at the preconditions. This could identify missing and unnecessary preconditions. It could also allow the domain expert to identify situations where the simulation behaves in an undesirable manner.

Reference List

[A$^+$95] John R. Anderson et al. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995.

[AFCFG97] A. Arruarte, I. Fernádez-Castro, B. Ferrero, and J. Greer. The IRIS shell: "how to build ITSs from pedagogical and design requisites". *International Journal of Artificial Intelligence in Education*, 8:341–348, 1997.

[AIS88] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *AAAI 1988*, pages 735–740, 1988.

[AJR97] Richard Angros, Jr, W. Lewis Johnson, and Jeff Rickel. Agents that learn to instruct. In *AAAI 1997 Fall Symposium Series: Intelligent Tutoring System Authoring Tools*, pages 1–8. AAAI Press, November 1997. Technical Report FS-97-01.

[Ana83] J. Anania. The influence of instructional conditions on student learning and achievement. *Evaluation in Education*, 7:1–92, 1983.

[And85] Peter Merrett Andreae. *Justified Generalization: Acquiring Procedures From Examples.* PhD thesis, MIT, 1985.

[Ang87a] D. Angluin. Learning regular sets from queries and counter-examples. *Information and Computation*, 75(2):87–106, 1987.

[Ang87b] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[Bal93] Cecile Balkanski. *Actions, Beliefs and Intensions in Multi-Action Utterances.* PhD thesis, Harvard University, May 1993.

[Bau98] Mathias Bauer. Acquisition of abstract plan descriptions for plan recognition. In *Fifteenth National Conference on Artificial Intelligence*, pages 936–941, Madison, WIsconsin, July 1998. The AAAI Press / The MIT Press.

[Bel98] Benjamin Bell. Investigate and decide learning environments: Specializing task models for authoring tool design. *The Journal of the Learning Sciences*, 7(1):65–105, 1998.

[Ben95] Scott Benson. Inductive learning of reactive action models. *Machine Learning: Proceedings of the 12th International Conference*, pages 47–54, 1995.

[Ble97] Stephen B. Blessing. A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education*, 8, 1997.

[Blo84] B. S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as ono-to-one tutoring. *Educational Reseacher*, pages 4–16, June/July 1984.

[BN95] Scott Benson and Nils J. Nilsson. Reacting, planning, and learning in an autonomous agent. In Koichi Furakawa, Donald Michie, and Stephen Muggleton, editors, *Machine Intelligence*, volume 14, pages 29–64. Oxford University Press, 1995.

[Boo85] J. H. Boose. A knowledge acquisition program for expert systems based on personal construct psychology. *International Journal of Man-Machine Studies*, 23(5):495–525, 1985.

[BS93] Michael S. Bocionek and Siegfried B. Sassin. Dialog-based learning (DBL) for adaptive interface agents and programming-by-demonstration systems. Technical Report CMU-CS-93-175, School of Computer Science, Carnegie Mellon University, July 1993.

[BSP85] Alan Bundy, Bernard Silver, and Dave Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27:137–181, 1985.

[Bur83] A. J. Burke. *Students' potential for learning contrasted under tutorial and group approaches to instruction.* PhD thesis, University of Chicago, 1983.

[BV96] Alberto Del Bimbo and Enrico Viario. Visual programming of virtual worlds animation. *IEEE Multimedia*, 3(1), 1996.

[C+93] Allen Cypher et al., editors. *Watch What I Do: Programming by Demonstration.* The MIT Press, 1993.

[Car70] J.R. Carbonell. AI in CAI: an artificial intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, 11(4):190–202, 1970.

[CBL+89] M. T. H. Chi, M. Bassok, M. W. Lewis, P. Reimann, and R. Glaser. Self-explanations: How students study and use examples to solve problems. *Cognitive Science*, 13:145–182, 1989.

[CBN89] Allan Collins, John Seely Brown, and Susan E. Newman. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. In Lauren B. Resnick, editor, *Knowing, learning, and instruction: essays in honor or Robert Glaser*, pages 453–494. L. Erlbaum Associates, Hillsdale, N.J., 1989.

[CG90] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In Yves Kodratoff and Ryszard S. Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 191–213. Morgan Kaufmann, San Mateo, CA, 1990.

[Chi97] Michelene T. H. Chi. Quantifying qualitative analysis of verbal data: A practical guide. *The Journal of the Learning Sciences*, 6(3):271–315, 1997.

[CKM93] Allen Cypher, David S. Kosbie, and David Maulsby. Characterizing PBD systems. In Allen Cypher et al., editors, *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993.

[Cla86] William J. Clancey. From GUIDON to NEOMYCIN and HERACLES in twenty short lessons: ORN final report 1979–1985. *AI Magazine*, 7(3):40–60, August 1986.

[CLCL94] Michelene T. H. Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian LaVancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18:439–477, 1994.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[Coh92] Philip R. Cohen. The role of natural language in a multimodel interface. In *UIST'92*, pages 143–149, Monterey California, 1992.

[CS95] Allen Cypher and David Canfield Smith. KIDSIM: End user programming of simulations. In *SIGCHI '95*, pages 27–34, Denver Colorado, May 1995. ACM SIGCHI.

[CV91] Michelene T. H. Chi and Kurt A. VanLehn. The content of physics self-explanations. *The Journal of the Learning Sciences*, 1(1):69–105, 1991.

[Dav84] Randall Davis. Interactive transfer of expertise. In Bruce G. Buchanan and Edward H. Shortliffe, editors, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, pages 171–205. Addison-Wesley Publishing Company, 1984.

[DHP$^+$94] Judy Delin, Anthony Hartley, Cecile Paris, Donia Scott, and Keith Vander Linden. Expressing procedural relationships in multilingual instructions. In *Proceedings of the Seventh International Workshop on Natural Language Generation*, pages 61–70, Kennebunkport, ME, 1994.

[DHW94] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 31–36, Chicago, Illinois, 1994. AAAI Press.

[Di 94] Barbara Di Eugenio. Action representation for interpreting purpose clauses in natural language instructions. In *Proceedings of the Fourth International Conference on Knowledge Representation and Reasoning*, 1994.

[DM86] Gerald DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[DR98] Wolff Daniel Dobson and Christopher K. Riesbeck. Tools for incremental development of educational software interfaces. In *CHI 98*, pages 384–391, Los Angles, CA, 1998.

[EEMT87] Larry Eshelman, Damien Ehret, John McDermott, and Ming Tan. MOLE: a tenacious knowledge-acquisition tool. *Int. J. of Man-Machine Studies*, 26:41–54, 1987.

[ES84] K. A. Ericsson and H. Simon. *Protocol Analysis: Verbal reports as data.* MIT Press, Cambridge, MA, 1984.

[Fel72] Katherine Voerwerk Feldman. The effects of the number of positive and negative instances, concept definitions, and emphasis of relevant attributes on the attainment of mathematical concepts. In *Proceedings of the Annual Meeting of the American Educational Research Association*, Chicago, Illinois, 1972.

[FMD$^+$96] H. Friedrich, S. Münch, R. Dillman, S. Bocionek, and M. Sassin. Robot programming by demonstration (RPD): Supporting the induction by human interaction. *Machine Learning*, 23:163–189, 1996.

[Gai87] Brian R. Gains. An overview of knowledge-acquisition and transfer. *Int. J. Man-Machine Studies*, 26:453–472, 1987.

[Gal90] Deborah Krawczak Galdes. *An Empirical study of Human Tutors: The Implications for Intelligent Tutoring Systems.* PhD thesis, The Ohio State University, 1990.

[GBW88] R. M. Gangé, L. J. Briggs, and Wager W. W. *Principles of Instructional Design.* Holt, Rinehart and Winston, third edition, 1988.

[GCV98] Abigail S. Gertner, Cristina Conati, and Kurt VanLehn. Procedural help in andes: Generating hints using a bayesian network student model. In *Fifteenth National Conference on Artificial Intelligence (AAAI 1998)*, pages 106–111, Madison, Wisconson, 1998.

[Gil92] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, 1992.

[GMAB93] S. Goldin-Meadow, M. W. Alibali, and R. Breckinridge Church. Transitions in concept acquisition: Using the hand to read the mind. *Psychological Review*, 100:279–298, 1993.

[Gru89] Thomas R. Gruber. Automated knowledge acquisition for strategic knowledge. *Machine Learning*, 4:293–336, 1989.

[Ham89] Kristian J. Hammond. CHEF. In C. Reisbeck and R. Shank, editors, *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[Hau88] David Haussler. Quantifying inductive bias: Artificial intelligence learning algorithms and valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[Hei89] Rosanna Heise. Demonstration instead of of programming: focussing attention in robot task acquisition. Technical Report Research Report No. 89/360/22, University of Calgary, September 1989.

[Hei93] Rosanna Heise. Programming robots by example. *International Journal of Intelligent Systems*, 8:685–709, 1993.

[HHR99] Patricia Y. Hsieh, Henry M. Halff, and Carol L. Redfield. Four easy pieces: Development systems for knowledge-based generative instruction. *International Journal of Artificial Intelligence in Education*, 10, 1999.

[Hil94] Randall W. Hill, Jr. Impasse-driven tutoring for reactive skill acquisition. Technical Report JPL Publication 94–9, Jet Propulsion Laboratory, California Institute of Technology, April 1994. Reprint of University of Southern California PhD thesis.

[HL93] Scott B. Huffman and John E. Laird. Learning procedures from interactive natural language instructions. In P. Utgoff, editor, *Machine Learning: Proceedings of the Tenth International Conference*, volume 15, page : a total of 12. ?, Amhearst, Mass., June 1993.

[HL95] Scott B. Huffman and John E. Laird. Flexibly instructable agents. *Journal of Artificial Intelligence Research*, 3:271–324, 1995.

[HMD73] John C. Houtz, J. William Moore, and J. Kent Davis. Effects of different types of positive and negative examples in learning "non-dimensioned" concepts. *Journal of Educational Psychology*, 64(2):206–211, 1973.

[HMP97] Haym Hirsh, Nina Mishra, and Leonard Pitt. Version spaces without boundary sets. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 491–496. AAAI Press/The MIT Press, 1997.

[Hof87] Robert R. Hoffman. The problem of extracting the knowledge of experts from the perspective of experimental psychology. *AI Magazine*, 8(2):53–67, 1987.

[HS91] David Hume and Claude Sammut. Using inverse resolution to learn relations from experiments. In *Proceedings of the Eighth Machine Learning Workshop*, Evanston, Il, July 1991.

[Huf94] Scott B. Huffman. *Instructable Autonomous Agents.* PhD thesis, University of Michigan, 1994.

[JH95] B. Jordan and A. Henderson. Interaction analysis: Foundations and practice. *The Journal of the Learning Sciences*, 4:39–103, 1995.

[JK97] Menachem Jona and Alex Kass. A full-integrated approach to authoring learning environments: Case studies and lessons learned. In *AAAI 1997 Fall Symposium Series: Intelligent Tutoring System Authoring Tools*, pages 39–43. AAAI Press, November 1997. Technical Report FS-97-01.

[JRSM98] W. Lewis Johnson, Jeff Rickel, R. Stiles, and Allen Munro. Integrating pedagogical agents into virtual environments. *Presence: Teleoperators and Virtual Environments*, 7(6):523–546, December 1998.

[Kel55] G. A. Kelly. *The psychology of personal constructs.* Norton, New York, 1955.

[KF93] David Kurlander and Steven Feiner. A history of editable graphical histories. In Allen Cypher et al., editors, *Watch What I Do: Programming by Demonstration*, pages 405–413. The MIT Press, 1993.

[KGF74] Herbert J. Klausmeier, E. S. Ghatala, and D. A. Frayer. *Conceptual Learning and Development, a Cognitive View.* Academic Press, New York, 1974.

[KM93] David S. Kosbie and Brad A. Myers. A system-wide macro facility based on aggregate events: A proposal. In Allen Cypher et al., editors, *Watch What I Do: Programming by Demonstration*, pages 433–444. The MIT Press, 1993.

[Kri95] Balachander Krishnamurthy, editor. *Practical Resusable UNIX Software.* John Wiley & Sons, New York, NY, 1995.

[KW88] Brent J. Krawchuk and Ian H. Witten. On asking the right questions. In *5th International Machine Learning Conference*, pages 15–21. Morgan Kaufmann, June 1988.

[Lan80] P. Langley. Finding common paths as a learning mechanism. In *Third Conference of the Canadian Society for Computational Studies of Intelligence*, pages 12–19, 1980.

[Lew92] John D. Lewis. Task acquisition from instruction. Master's thesis, University of Calgary, 1992.

[Lie94] Henry Lieberman. A user interface for knowledge acquisition form video. In *Twelfth National Conference of the American Association for Artificial Intelligence*, August 1994.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[LW99] Tessa A. Lau and Daniel S. Weld. Programming by demonstration: An inductive learning formulation. In *1999 International Conference on Intelligent User Interfaces*, pages 145–152, Redondo Beach, CA, January 1999.

[MA97] Nigel Major and Shaaron Ainsworth. Developing intelligent tutoring systems using a psychologically motivated authoring environment. In *AAAI 1997 Fall Symposium Series: Intelligent Tutoring System Authoring Tools*, pages 53–59. AAAI Press, November 1997. Technical Report FS-97-01.

[Maj95] Nigel Major. Modeling teaching strategies. *Journal of Artificial Intelligence in Education*, 6(2/3):117–152, 1995.

[Mau94] David Maulsby. *Instructable Agents*. PhD thesis, University of Calgary, June 1994.

[MAW97] Nigel Major, Shaaron Ainsworth, and David Wood. REDEEM: Exploiting the symbiosis between psychology and authoring environments. *International Journal of Artificial Intelligence in Education*, 8:317–340, 1997.

[ME89] Chris Mellish and Roger Evans. Natural language generation from plans. *Computational Linguistics*, 15(4), 1989.

[Mit78] Tom M. Mitchell. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Stanford University, 1978.

[Mit82] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Mit93] Vibhu O. Mittal. Generating natural language descriptions with integrated text and examples. Technical Report ISI/RR-93-392, USC/Information Sciences Institute, September 1993.

[MJP$^+$97] Allen Munro, Mark C. Johnson, Quentin A. Pizzini, David S. Surmon, Douglas M. Towne, and James L. Wogulis. Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education*, 8:284–316, 1997.

[MJSW93] A. Munro, M. C. Johnson, D. S. Surmon, and J. L. Wogulis. Attribute-centered simulation authoring for instruction. In *Proceedings of the AI-ED 93 World Conference of Artificial Intelligence in Education*, pages 82–89, Edinburgh, Scotland, 1993.

[MKKC86] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[MMS90] Tom M. Mitchell, Sridbar Mabadevan, and Louis I. Steinberg. LEAP: A learning apprentice for VLSI design. In *Machine Learning An Artificial Intelligence Approach*, volume III, pages 271–289. Morgan Kaufmann, San Mateo, CA, 1990.

[MP93] Vibhu O. Mittal and Cecile L. Paris. Generating natural language descriptions with examples: Differences between introductory and advanced texts. In *Proceedings of the Eleventh National Conference on on Artificial Intelligence*, pages 271–276, Washington, DC, July 1993.

[MR91] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 634–639, Menlo Park, CA, 1991. AAAI Press.

[MT69] S. M. Markle and P. W. Tiemann. *Really Understanding Concepts.* Stipes Press, Urbana, Illinois, 1969.

[MUB83] Tom M. Mitchell, Paul E. Utgoff, and Ranan Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning An Artificial Intelligence Approach*, volume I. Morgan Kaufmann, San Mateo, CA, 1983.

[Mur97] Tom Murray. Expanding the knowledge acquisition bottleneck for intelligent tutoring systems. *International Journal of Artificial Intelligence in Education*, 8:222–232, 1997.

[Mur98] Tom Murray. Authoring knowledge-based tutors: Tools for content, instructional strategy, student model, and interface design. *The Journal of the Learning Sciences*, 7(1):5–64, 1998.

[Mus93] Mark A. Musen. An overview of knowledge acquisition. In J. M. David, J. P. Krivine, and R. Simmons, editors, *Second Generation Expert Systems*, pages 405–427. Springer-Verlag, 1993.

[MW93] David Maulsby and Ian H. Witten. Metamouse: An instructible agent for programming by demonstration. In *What What I Do: Programming by Demonstration.* The MIT Press, 1993.

[MWM94] Antonija Mitrović, Ian H. Witten, and David L. Maulsby. An experiment in the application of similarity-based learning to programming by example. *International Journal of Intelligent Systems*, 9:341–364, 1994.

[New90] Allen Newell. *Unified Theories of Cognition.* Harvard University Press, 1990.

[Nor88] Donald A. Norman. *The Psychology of Everyday Things.* Basic Books, New York, 1988.

[OC96] Tim Oates and Paul R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 863–868, 1996.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[Pea96] Douglas John Pearson. *Learning Procedural Planning Knowledge in Complex Environments.* PhD thesis, University of Michigan, 1996.

[PK86] Bruce W. Porter and Dennis F. Kibler. Experimental goal regression: A method for learning problem-solving heuristics. *Machine Learning*, 1:249–286, 1986.

[PL96] Douglas J. Pearson and John E. Laird. Toward incremental knowledge correction for agents in complex environments. In S. Muggleton, D. Michie, and K. Furukawa, editors, *Machine Intelligence*, volume 15. Oxford University Press, 1996.

[Pol90] Martha Pollack. Plans as complex mental attitudes. In Phil Cohen, Jerry Morgan, and Martha Pollack, editors, *Intention in Communication.* MIT Press, 1990.

[PS92] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, College Park, Maryland, 1992. Morgan Kaufmann.

[PV96] Cecile Paris and Keith Vander Linden. An interactive support tool for writing multilingual manuals. *IEEE Computer*, 29(7):49–56, 1996.

[PVF$^+$95] Cecile Paris, Keith Vander Linden, Markus Fischer, Anthony Hartley, Lyn Pemberton, Richard Power, and Donia Scott. A support tool for writting multilingual instructions. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1398–1404, Montreal, Canada, 1995.

[RB98] Steven Ritter and Stephen B. Blessing. Authoring tools for component-based learning environments. *The Journal of the Learning Sciences*, 7(1):107–132, 1998.

[Red92] Michael A. Redmond. *Learning by observing and understanding expert problem solving.* PhD thesis, Georgia Institute of Technology, 1992.

[Red97] Carol Luckhardt Redfield. An ITS authoring tool: Experimental advanced instructional design advisor. In *AAAI 1997 Fall Symposium Series: Intelligent Tutoring System Authoring Tools*, pages 72–78. AAAI Press, November 1997. Technical Report FS-97-01.

[Ren97] Alexander Renkl. Learning from worked-out examples: A study on individual differences. *Cognitive Science*, 21(1):1–29, 1997.

[Ric89] Jeff W. Rickel. Intelligent computer-aided instruction: A survey organized around system components. *IEEE Transactions on Systems, Man and Cybernetics*, 19(1):40–57, 1989.

[RJ99] J. Rickel and W. L. Johnson. Animated agents for procedural training in virtual reality: Perception, cognition, and motor control. *Applied Artificial Intelligence*, 1999.

[RK96] Steven Ritter and Kenneth R. Koedinger. An architecture for plug-in tutor agents. *Journal of Artificial Intelligence in Education*, 7(3/4):315–347, 1996.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach.* Prentice Hall Series in artificial intelligence. Prentice Hall, 1995.

[RS90] Ronald L. Rivest and Robert E. Schapire. A new approach to unsupervised learning in deterministic environments. In Yves Kodratoff and Ryszard S. Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 670–684. Morgan Kaufmann, San Mateo, CA, 1990.

[RS97] Charles Rich and Candace L. Sidner. COLLAGEN: When agents collaborate with people. In *Proceedings of the First International Conference on Autonomous Agents*, pages 284–291, February 1997.

[SB86] Claude Sammut and Ranan B. Banerji. Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume II, pages 167–191. Morgan Kaufmann, Los Altos, CA, 1986.

[Sch53] H. Scheffé. A method for judging all contrasts in the analysis of variance. *Biometrika*, 40:87–104, 1953.

[Sch94] Roger C. Schank. Goal-based scenarios: A radical look at education. *The Journal of the Learning Sciences*, 4(3):429–453, 1994.

[SCS94] David Canfield Smith, Allen Cypher, and Jim Spoher. KIDSIM: Programming agents without a programming language. *CACM*, 94(7):55–67, July 1994.

[Seg87] Alberto Maria Segre. A learning apprentice system for mechanical assembly. In *IEEE Third Conference on Artificial Intelligence Applications*, pages 112–117, 1987.

[Sel74] J.A. Self. Student models in computer-aided instruction. *International Journal of Man-Machine Studies*, 6:261–276, 1974.

[SG88] Mildred L. G. Shaw and Brian R. Gaines. An interactive knowledge-elicitation technique using personal construct technology. In *Knowledge Acquisition for Expert Systems: A Practical Handbook*, pages 109–136. Plenum Press, New York, 1988.

[She93] Wei-Min Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 11(4):250–265, 1993.

[She94] Wei-Min Shen. *Autonomous Learning From The Environment.* W. H. Freeman, New York, NY, 1994.

[She97] Michael Shermer. *Why People Believe Wierd Things.* W. H. Freeman and Company, New York, NY, 1997.

[SJ91] Roger C. Schank and Menachem Y. Jona. Empowering the student: New perspectives on the design of teaching systems. *The Journal of the Learning Sciences*, 1(1):7–35, 1991.

[SMP95] Randy Stiles, Laurie McCarthy, and Michael Pontecorvo. Training studio: A virtual environment for training. In *Workshop on Simulation and Interaction in Virtual Environments (SIVE-95)*, Iowa City, IW, July 1995. ACM Press.

[SR90] Benjamin D. Smith and Paul S. Rosenbloom. Incremental non-backtracking focusing: A polynomial bounded generalization algorithm for version spaces. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 848–853, 1990.

[SS98] Raymund Sison and Masamichi Shimura. Student modeling and machine learning. *International Journal of Artificial Intelligence in Education*, 9:128–158, 1998.

[Tec92] Gheorghe Tecuci. Automating knowledge acquisition as extending, updating, and improving a knowledge base. *IEEE Transactions on Systems, Man and Cybernetics*, 22(6):1444–1460, 1992.

[TH96] Gheorghe Tecuci and Michael R. Hieb. Teaching intelligent agents: the Disciple approach. *International Journal of Human-Computer Interaction*, 8(3):259–285, 1996.

[THD95] Gheorghe Tecuci, Michael R. Hieb, and Tomasz Dybala. Building an adaptive agent to monitor and repair the electrical power system of an orbital satellite. In *Goddard Conference on Space Applications of Artificial Intelligence and Emerging Information Technologies*, pages 57–71, NASA Goddard, Greenbelt, Maryland, 1995.

[TK90] Gheorghe Tecuci and Yves Kodratoff. Apprenticeship learning in imperfect domain theories. In *Machine Learning An Artificial Intelligence Approach*, volume III, pages 514–552. Morgan Kaufmann, San Mateo, CA, 1990.

[TK98] Gheorgie Tecuci and Harry Keeling. Delevoping intelligent educational agents with the Disciple learning agent shell. In Barry P. Goettl, Henry M. Halff, Carol L. Redfield, and Valerie J. Shute, editors, *Intelligent Tutoring Systems: 4th international conference*, pages 454–463. Springer-Verlag, Berlin, 1998.

[Tow97a] Douglas M. Towne. Approximate reasoning techniques for intelligent diagnostic instruction. *International Journal of Artificial Intelligence in Education*, 8:262–283, 1997.

[Tow97b] Douglas M. Towne. Diagnostic tutoring using qualitative symptom information. In *AAAI 1997 Fall Symposium Series: Intelligent Tutoring System Authoring Tools*, pages 86–95. AAAI Press, November 1997. Technical Report FS-97-01.

[Utg86] Paul E. Utgoff. shift of bias for inductive concept learning. In *Machine Learning An Artificial Intelligence Approach*, volume II, pages 107–148. Morgan Kaufmann, Los Altos, CA, 1986.

[Van83] Kurt VanLehn. Felicity conditions for human skill acquisition: Validating an AI-based theory. Research report no. CIS-21, Xerox Palo Alto Research Center, 1983.

[Van87] Kurt VanLehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31:1–40, 1987.

[Van93] Keith Vander Linden. *Speaking of Actions: Choosing Rhetorical Status and Grammatical Form in Instructional Text Generation*. PhD thesis, University of Colorado, Department of Computer Science, 1993.

[Van99] Kurt VanLehn. Rule-learning events in the acquisition of a complex skill: An evaluation of Cascade. *The Journal of the Learning Sciences*, 8(1):71–125, 1999.

[VCP+95] Manuela Veloso, Jaime G. Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), January 1995.

[VD96] W.R. VanJoolingen and T. DeJong. Design and implementation of simulation-based discovery environments: the SMISLE solution. *International Journal of Artificial Intelligence in Education*, 7:253–276, 1996.

[VJC92] Kurt VanLehn, Randolph M. Jones, and Michelene T. H. Chi. A model of the self-explanation effect. *The Journal of the Learning Sciences*, 2(1):1–59, 1992.

[VM95] Keith Vander Linden and James H. Martin. Expressing rhetorical relations in instructional test: A case study of the purpose relation. *Computational Linguistics*, 21:29–57, March 1995.

[Wan95] Xuemai Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *The 12th International Conference on Machine Learning*, 1995.

[Wan96a] Xuemai Wang. A multistrategy learning system for planning operator acquisition. In *The Third International Workshop on Multistrategy Learning*, Harpers Ferry, West Virginia, May 1996.

[Wan96b] Xuemai Wang. Planning while learning operators. In *The Third International conference on artificial planning systems*, May 1996.

[Wan96c] Xuemei Wang. *Learning Planning Operators by Observation and Practice*. PhD thesis, Carnegie Mellon University, 1996.

[WC87] Beverly Woolf and Patricia A. Cunningham. Multiple knowledge sources in intelligent teaching systems. *IEEE Expert*, 2(2):41–54, 1987.

[Wd90] Daniel S. Weld and Johan de Kleer, editors. *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufman, San Mateo, CA, 1990.

[Wel94] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, pages 27–61, Winter 1994.

[Wen87] Etienne Wenger. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to Communication of Knowledge.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[Wil90] David C. Wilkins. Knowledge base refinement as improving an incorrect and incomplete domain theory. In *Machine Learning An Artificial Intelligence Approach*, volume III, pages 493–513. Morgan Kaufmann, San Mateo, CA, 1990.

[WW72] Thomas H. Wonnacott and Ronald J. Wonnacott. *Introductory Statistics for Business and Economics.* John Wiley & Sons, 1972.

[You97] R. Michael Young. *Generating Descriptions of Complex Activities.* PhD thesis, University of Pittsburgh, 1997.

[YPL77] Richard M. Young, Gordon D. Plotkin, and Reinhard F. Linz. Analysis of an extended concept-learning task. In *Proceedings of the Fifth International Conference on Artificial Intelligence*, page 285, 1977.

Appendix A

Implementation

A.1 Architecture

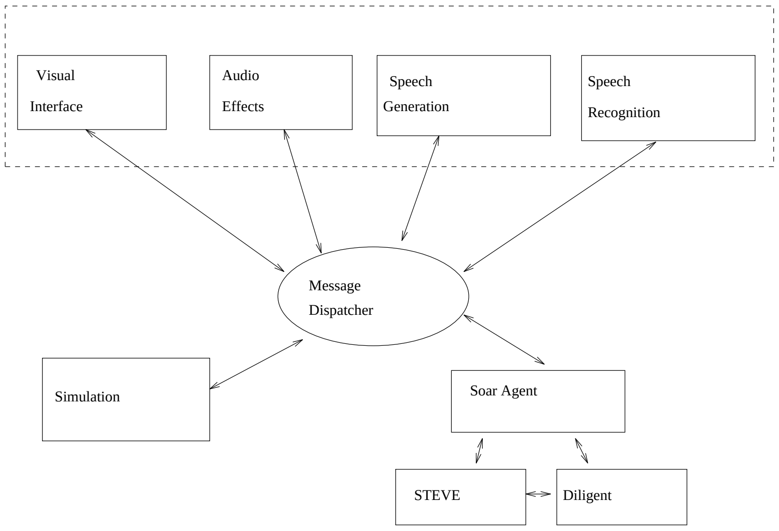


Figure A.1: The VET Software Architecture

Diligent was implemented in the context of the Virtual Environments for Training (VET) project [JRSM98]. For purposes of modularity, the different components run as separate processes on possibly different machines.

The project uses Silicon Graphics workstations running version 6.2 – 6.5 of the IRIX operating system. Figure A.1 shows a schematic of the VET architecture.

Message Dispatcher. The software components talk to each other via the message dispatcher. For this we are using Sun's ToolTalk^{TM}.

Visual Interface. The visual interface is the graphical representation of the environment. The visual interface is provided by Lockheed Martin's Vista Viewer [SMP95]. On the VET project, two types of visual interfaces are supported: a browser on the computer console and an immersive virtual reality environment that uses a head-mounted display and data gloves. Because of the need to use the keyboard and to interact with Diligent's menus, Diligent only supports authoring with the browser. However, once a procedure has been authored, the procedure can be used to teach students with either the browser or the immersive environment.

Audio Effects. Human students have the ability to hear various sound effects on their head-mounted display. Diligent does not deal with this capability.

Speech Generation. STEVE is able to speak to students. Diligent's test subjects used this capability when testing procedures. This capability is provided by Entropic's TrueTalk^{TM}.

Speech Recognition. This component is allows students to communicate with STEVE agents. The capability is provided by Entropic's GrapHVite^{TM}. Diligent does not deal with actions that involve communication.

Simulation. The simulation controls the environment. It is implemented with VIVIDS, which is a version of RIDES [MJP+97]. RIDES was developed at the USC Behavior Technology Laboratory (BTL). The people at BTL modified VIVIDS so that Diligent was able to save and restore environment configurations.

Soar Agent. The Soar agent [LNR87] is a production system that contains both the **STEVE**[1] tutor [RJ99] and the **Diligent** authoring program. STEVE and Diligent are separate modules that behave like separate programs.

STEVE uses a synthetic body (Figure A.2) to interact with students. STEVE uses the body to perform activities such as demonstrating procedures and pointing to or looking at objects. STEVE is primarily implemented as Soar productions. STEVE uses tksoar version 7.0.0.beta, TCL version 7.4 and TK version 4.0.

Diligent is primarily implemented in TCL/TK [Ous94]. Most of Diligent resides in the same process as STEVE, but the code that produces graphs of procedures has its own process. The graph process uses the tkdot portion of the Graph Visualization tools from AT&T Laboratories and Bell Laboratories (Lucent Technology) [Kri95]. The graph process uses TCL version 7.6, TK version 4.2 and the TK Dash patch by Jan Nijtmans.

A.2 Maintenance of Agenda

One of problems faced by a system like Diligent is properly sequencing its input. Input can come from both the environment and the instructor. Furthermore, the environment and the instructor can be sending input at the same time. Additionally, some activities may involve a sequence of behaviors, some of which can take variable amounts of time. For example, consider initializing the environment before the start of a procedure's second demonstration. The following activities take place.

[1]Soar Training Expert for Virtual Environments

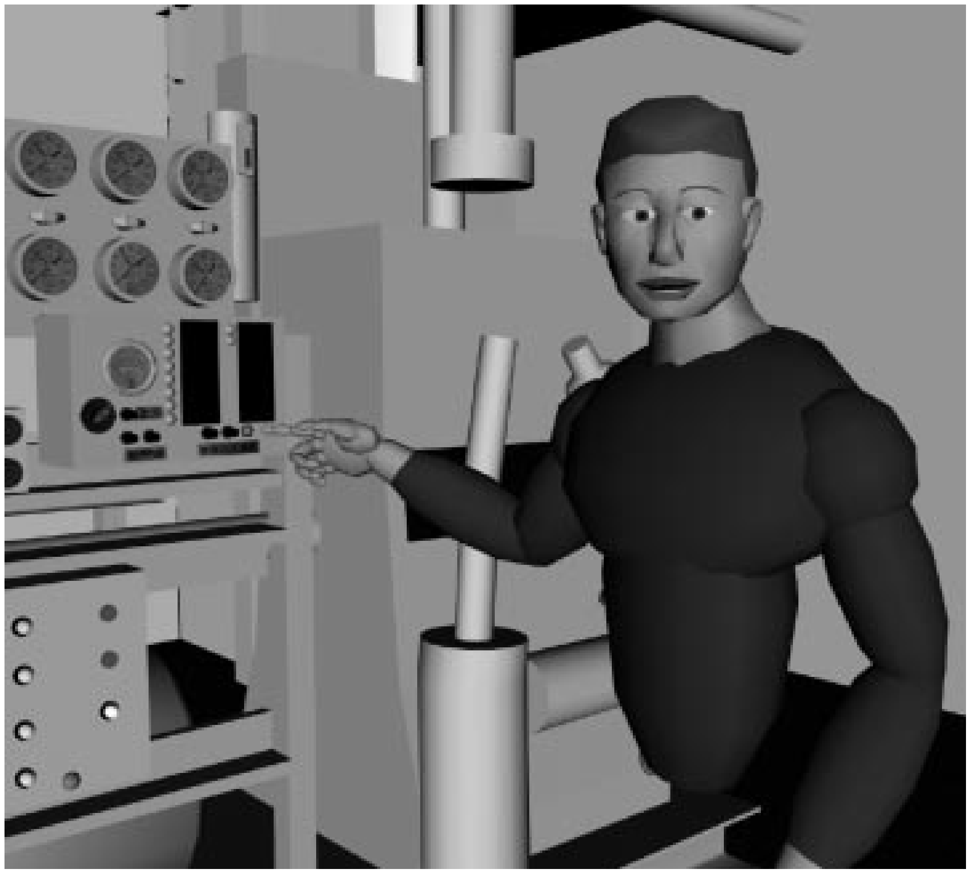


Figure A.2: The STEVE Tutoring Agent

1. The instructor is asked for an initial environment configuration. (Assume that the configuration matches the first configuration.)

2. The environment is reset.

3. At this point, Diligent may have records of actions performed in the environment that have not yet been processed. In order to prevent confusion, Diligent deletes these records.

4. Actions in the path's prefix are replayed.

5. The instructor can now add additional actions to the prefix.

6. The instructor indicates that the demonstration should start.

Because we want the instructor have maximum flexibility for interleaving activities, it is inappropriate to encode a fixed sequence of activities in a procedure or grammar. An approach is needed that allows maximum flexibility and minimizes the code that handles special cases.

To solve this problem, Diligent manages the interaction with an agenda. Diligent's *agenda* has stack of lists. Each level in the stack corresponds to one procedure, and each list contains activities to perform for that procedure. The dialog with the instructor is focused on the procedure at the top of the stack. To prevent confusion and to avoid problems, some activities are only allowed on the top procedure. The restricted activities include demonstrations, experiments and using STEVE to test the procedure.[2]

This approach was influenced by other work. The idea of a stack of procedures where the agent focuses on the top procedure was borrowed from Instructo-Soar [HL95]. The idea for each level of the agenda to contain a list of activities was inspired by COLLAGEN [RS97], whose agenda is a stack of plans for managing user interaction.

A.3 Providing Feedback About Diligent's Beliefs

Another problem faced by Diligent is providing feedback about its confidence in aspects of its knowledge base (e.g. how certain is Diligent that a causal link is correct?). By providing feedback, Diligent indicates what it believes strongly as well as areas of uncertainty where the instructor could focus.

To compute its confidence, Diligent could have used a formal numeric approach, such as certainty factors, fuzzy logic or Dempster-Scafer theory [RN95]. However, Diligent may have little data from which to calculate numeric values. Furthermore, it was hypothesized that numeric values might confuse users who are not experts in this area.

Instead, Diligent uses a small set of symbolic status values to describe its beliefs. These values are not described earlier because they are a minor part of the system and are not based on a rigorous theory. Nevertheless, the status values are important for two reasons.

- The status values are used in the user interface, which is described in Appendix D.

[2]Using an agenda greatly reduced the user interface's complexity.

- The status values are useful when using multiple paths to generate a plan.[3]

Status value	**Is the object used in a plan?**	**Meaning**
required	yes	Instructor has indicated that it be used.
suspect	yes	Instructor has indicated that it be used, but he appears to have made a mistake. (The object will still be used.)
provisional	yes	Likely to be correct.
ignored	no	Likely to be correct but not needed.
unlikely	no	Appears to be incorrect.
useless	no	Evidence strongly suggests it to be incorrect.
rejected	no	Instructor has indicated that it should not be used.

Table A.1: Status Values Used by Diligent

The status values used by Diligent are shown in Table A.1. The types of objects that have status values are preconditions, goal conditions, causal links and ordering constraints. By default, Diligent gives a status value of provisional to objects that it believes to be needed. The status values required and rejected are only used when the instructor explicitly indicates whether or not that object should be used when building a plan. The status value ignored is only used with ordering constraints involving a step that represents the procedure's initial state or goal state.

These status values are similar in concept to the three sets used to contain operator preconditions (i.e. s-rep, h-rep and g-rep) but are not the same; preconditions in the h-rep and g-rep have a status of provisional unless the instructor indicates that they should be required.

As mentioned earlier, the status values are useful when hypothesizing goal conditions using multiple paths. A useful heuristic is that a condition is a goal condition when the condition's attribute value changes during at least one path and the condition is present in the final state of every path. These hypothesized goal conditions are given a status of provisional. The heuristic also identifies conditions that appear to be goal conditions in some paths but not in others. These conditions have a status of unlikely or suspect. Conditions with a status of unlikely indicate that the instructor may have made an error in one of the paths, while conditions with a status of suspect indicate an error because a previously required goal condition is not satisfied in the final state of a least one path.

[3]The capability to generate a plan from multiple paths was removed from Diligent. Some of the issues are described in Section 8.4.1.1.

Appendix B

Evaluation Materials

This appendix contains material used for evaluating Diligent.

B.1 Background Questionnaire

The first thing subjects did was fill out this questionnaire.

Name:

Date:

1) Educational background. How many total years of
education do you have (e.g. 12 years of high school +
4 years of college + 2 years of graduate school)?
What degrees do you have and in what subjects? If you are
a graduate student, when did you start graduate school?

2) How old are you?
<25 <30 <35 <40 <50 >50

3) Are you male or female?

4) Are you color-blind? If so, in what way?

5) Are you right or left handed?

6) Do you have a personal computer at home?

7) Do you use a computer at work?

8) What is your occupation?
9) How many hours a week do you typically use a computer?

10) During a typical week, what are your primary activities
on a computer and how many hours do you spend on each?

programming:
word processing:
browsing:
using a spread sheet:
other (name the activities):

11) What are the main activities you have performed on a computer in the last week? About how many hours have you spent on each?
1.
2.
3.

12) Which programming languages do you have a lot of experience with?

13) How would you rate yourself as a computer programmer?
a) not a programmer
b) novice
c) intermediate
d) good
e) expert

14) Circle the following topics for which you feel that you have significant knowledge.
a) AI planning techniques
b) machine learning induction techniques
c) programming by demonstration
d) high pressure air compressor maintenance
e) machine maintenance in general
f) Diligent: the system we are testing

15) How would you rate your ability to read English?
a) poor
b) moderate
c) good
d) excellent
e) English is my first language

B.2 Procedure Representation Description

This section was read by subjects near the start of the first day's training. The subjects then filled out the worksheet on Diligent's procedure representation (Section B.3).

Note that the tutorial uses the term "ordering relationships" instead of the term "step relationships" that is used in this thesis.

In this section, we'll discuss how procedures are represented.

First, we need to define some terminology. The environment is represented by a set of attributes. Each attribute has a value. A *condition* contains an attribute and its value. A condition is true, or satisfied, when the attribute has the value and false when the attribute doesn't have the value.

A procedure transforms an initial environment state to a desired goal state. The state is transformed through a sequence of steps, where each step represents some action that is performed in the environment. A procedure is finished when all its goal conditions are true.

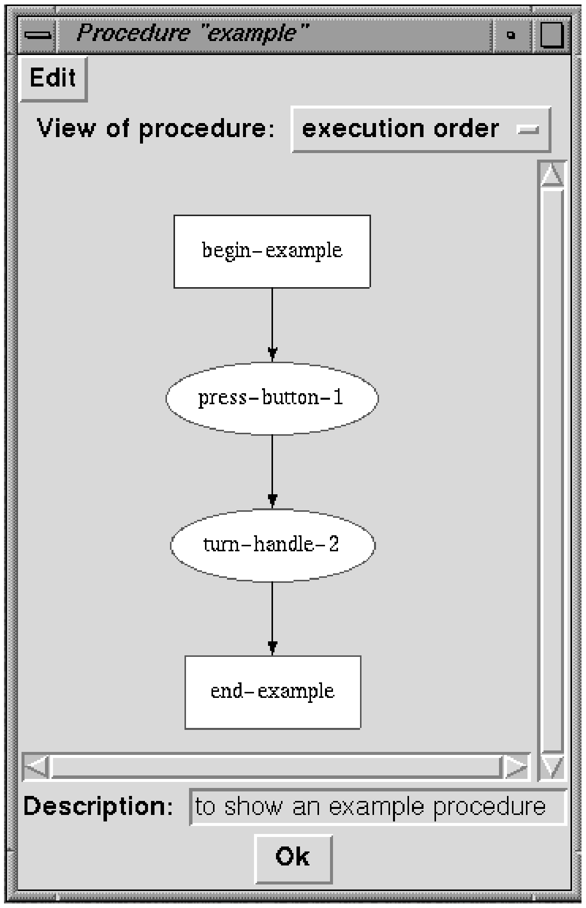
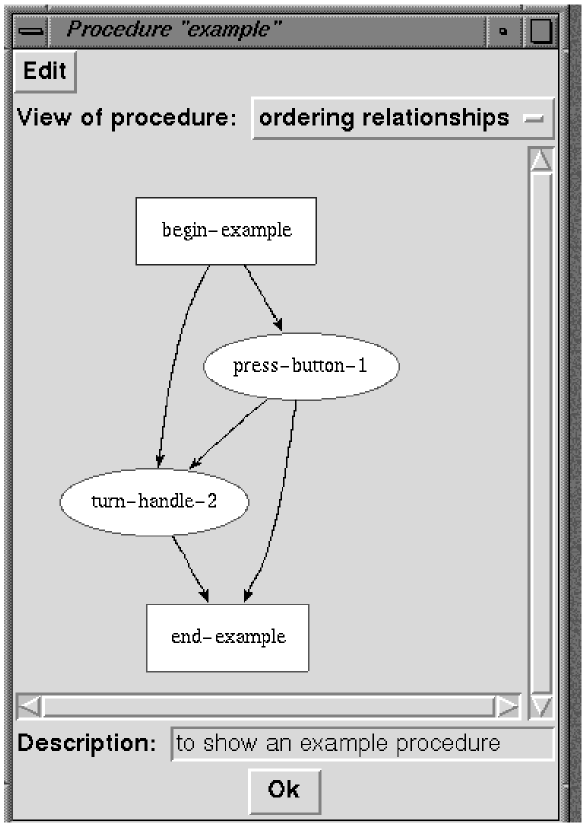


Figure B.1: Procedure with Steps in Specification Order

Figure B.1 shows a procedure called "example". The "begin-example" step represents the initial state, and the "end-example" step represents the goal state. The steps "press-button-1" and "turn-handle-2" represent the actions performed during the procedure. The steps are ordered by the sequence in which they were specified.

However, a procedure's steps don't have to be performed in the order that they were specified. Instead, the steps in a procedure can be performed in any order that satisfies the preconditions of each step.



View of procedure: ordering relationships

Description: to show an example procedure

Figure B.2: Procedure with Steps Ordered by Dependencies

Figure B.2 shows procedure “example” where the steps are ordered by dependencies of later steps on earlier steps.

In order to keep track of the preconditions and state changes of each step, every step is associated with an operator. An *operator* models an action performed in the environment. An operator can have multiple effects. Each *effect* has a set of preconditions and a set of state changes. If an effect’s preconditions are satisfied, the effect’s state changes will be observed.

Because operators model actions, each operator can be associated with multiple steps.

Because an operator can have multiple effects, each step is associated with a subset of an operator’s effects.

Operator: toggle-valve1

Action: toggling valve Valve1

Effect 1:	Effect 2:
Preconditions:	Preconditions:
(Valve1 = open)	(Valve1 = shut)
State changes:	State changes:
(Valve1 = shut)	(Valve1 = open)

Figure B.3: Operator with Two Effects

Figure B.3 shows an operator that models the toggling of valve Valve1. The operator has two effects: if the valve is open, it becomes shut (Effect 1), and if the valve is shut, it becomes open (Effect 2).

```
Procedure: Example2
    The specified order of steps: toggle-valve-1 → toggle-valve-2

Step: toggle-valve-1
    Operator: toggle-valve
    Operator Effects : Effect 1

    Step preconditions: (Valve1 = open)
    Step state changes: (Valve1 = shut)

Step: toggle-valve-2
    Operator: toggle-valve
    Operator Effects : Effect 2
    Step prerequisites: (Alarm-light1 on)

    Step preconditions: (Valve1 = shut)(Alarm-light1 = on)
    Step state changes: (Valve1 = open)
```

Figure B.4: Example Steps

Figure B.4 shows the steps in procedure Example2. Both steps use the operator in figure B.3. The first step (toggle-valve-1) shuts Valve1, and the second step (toggle-valve-2) opens Valve1. All preconditions of the first step (toggle-valve-1) come from the operator effect (Effect 1). Although the second step (toggle-valve-2) also gets preconditions from an operator effect (Effect 2), the second step has an additional precondition (Alarm-light1 = on). The additional precondition is step prerequisite. A *step prerequisite* is a precondition that belongs only to the step and not to the operator effects associated with the step.

A step prerequisite allows you to specify additional preconditions that are not required by the operator effects associated with the step.

Unfortunately, to actually perform a procedure, we need to know more that the preconditions of each step; we need to know the how the steps depend on each other. This involves knowing which steps establish preconditions of other steps. It also involves knowing if the state changes of one step will interfere with the preconditions of other steps.

Because the dependencies between steps contain the preconditions of each step, only the dependencies will be given to the Steve tutor.

Figure B.5 shows the dependencies between the steps in procedure Example2 (figure B.4). In this document, these dependencies will be called *ordering relationships* because they order a procedure's steps. Diligent uses two types of ordering relationships: causal links and ordering constraints.

A *causal link* is an attribute value caused by one step that is a precondition for a later step. Each step precondition can have a causal link. In the example, the first two causal links are actually dependencies on the procedure's initial state (begin-Example2). The

Procedure's initial state (begin-Example2):
(Valve1 = open)(Alarm-light1 = on)

Procedure goals (end-Example2):
(Valve1 = open)

Causal links:
begin-Example2 (Valve1 = open) toggle-valve-1
begin-Example2 (Alarm-light1 = on) toggle-valve-2
toggle-valve-1 (Valve1 = shut) toggle-valve-2
toggle-valve-2 (Valve1 = open) End-Example2

Ordering constraints:
toggle-valve-1 before toggle-valve-2

Figure B.5: Procedure Example2's Dependencies

last causal link is between the last step (toggle-valve-2) and the procedure's goal (end-Example2). A causal link with the procedure's goal indicates that the step establishes one of the procedure's goal conditions.

Causal links are used to represent the preconditions of steps and to provide explanations of how earlier steps affect later steps.

An *ordering constraint* indicates the relative order for performing a pair of steps. In the example, the first step (toggle-valve-1) should be performed before the second step (toggle-valve-2). There are no ordering constraints involving procedure's initial state (begin-Example2) and goals (end-Example2) because all steps are performed after the initial state and before the end of the procedure.

Ordering constraints are used to determine which step to perform when when all the preconditions of multiple steps are satisfied.

You may have noticed that the procedure's goal condition is satisfied in the initial state. This means that none of the procedure's steps would normally be performed. However, if the procedure was started when Valve1 was shut, then the second step would be performed.

Because understanding this chapter will prevent confusion, stop reading the tutorial. Please fill out the worksheet on the next page in your directions. When you are satisfied with with your answers, continue reading the tutorial and verify that your answers are correct.

B.3 The Procedure Representation Worksheet

After subjects read the tutorial chapter on the procedural representation, they filled out this questionnaire. When they were done, they checked their answers against those in section B.4.

In the following, circle the correct answers. (More than one answer may be correct.) If you discover that you've made an mistake, just change your answer.

1. Do the steps in a procedure change the state of the environment? True or False
2. A procedure is finished when
   (a) All its steps are executed
   (b) All its goal conditions are true (or satisfied)
3. Steps have to be performed in the order that they are specified? True or False
4. An operator models an action performed in the environment? True or False
5. Each step
   (a) Is Associated with only one operator
   (b) Can be associated with multiple operators
   (c) Is associated with only one operator effect
   (d) Can be associated with multiple operator effects.
6. Each operator
   (a) Is associated with an action performed in the environment
   (b) Is associated with multiple actions performed in the environment
   (c) Can have multiple effects
   (d) Is associated with a single step
   (e) Can be associated with multiple steps
7. Each operator effect
   (a) Has preconditions
   (b) Has state changes
   (c) Has causal links
   (d) Has ordering constraints
   (e) Produces the given state changes if the preconditions are satisfied
8. Step preconditions
   (a) Include all preconditions from the associated operator effects
   (b) Do not include the preconditions from the associated operator effects
   (c) Can include step specific preconditions called step prerequisites

9. Dependencies between steps
   (a) Are called ordering relationships
   (b) Include step preconditions
   (c) Include operator preconditions
   (d) Include causal links
   (e) Include ordering constraints

10. Causal links
   (a) Indicate that an earlier step establishes a precondition of a later step.
   (b) Indicate the relative order for performing a pair of steps
   (c) Are used to provide explanations about the dependencies between steps
   (d) Can involve more than two steps

11. Ordering constraints
   (a) Indicate that an earlier step establishes a precondition of a later step.
   (b) Indicate the relative order for performing a pair of steps
   (c) Are used to provide explanations about the dependencies between steps
   (d) Can involve more than two steps

12. You may want ordering constraints between a pair of steps when
   (a) There is a causal link between the steps
   (b) The state changes of the later step interfere with the preconditions of the earlier step
   (c) The later step is specified immediately after the first step

13. What is given to the Steve tutor?
   (a) A set of steps
   (b) A set of ordering relationships (i.e. causal links and ordering constraints)
   (c) A set of causal links that establish the procedure's goal conditions
   (d) A set of step preconditions
   (e) A set of operators

B.4 Worksheet Answers

These answers were contained in tutorial.

1. True.
2. (b).
3. False.
4. True.
5. (a),(d).
6. (a),(c),(e).
7. (a),(b),(e).
8. (a),(c).
9. (a),(d),(e).
10. (a),(c).
11. (b).
12. (a),(b).
13. (a),(b),(c).

Do you have any questions about these answers?

B.5 The Post-Test

The last thing that subjects did was answer the following questions.

How did you like it

In the following, please provide answers from 1 to 7. (1 means not at all, 4 means somewhat, and 7 means a great deal.) If you cannot answer a question write N/A.

The following questions were only given to subjects who only used an editor

Authoring

a) Did you like the system?
b) Was it easy to use?
c) Was it easy to specify a procedure's steps?
d) Was it easy to identify a step's preconditions?
e) Was it easy to identify a step's state changes?
f) Was it easy to identify how operators influenced causal links and ordering constraints?
g) Any other comments about authoring?

The following questions were only given to subjects who demonstrated.

Demonstrating

a) Did you like the system?
b) Was it easy to use?
c) Was it easy to demonstrate a procedure?
d) Did you find additional demonstrations useful?
e) Was it easy to specify a procedure's steps?
f) Was it easy to identify a step's preconditions?
g) Was it easy to identify a step's state changes?
h) Was it easy to identify how operators influenced causal links and ordering constraints?
i) Any comments about demonstrations?

The following questions were only given to subjects who experimented.

Experiments

a) Did you like experimenting?
b) Did experiments take too long?
c) Did experiments save you work?
d) Did experiments find errors that you would have missed?
e) Any comments about experiments?

Were there any other aspects of system that were useful or worth mentioning?

Thank you!

B.6 The Directions Given Subjects

This packet contains your directions for authoring procedures using Diligent.

Please go to the next page and answer the questions.

At this point, the subjects filled out the background questionnaire.

Please indicate that you are ready to continue.

First Day Directions

You will be given the Diligent tutorial.

Please open the tutorial and read through the first chapter and **stop when you've finished it. Indicate that you are done and ask to continue.**

Now work through the rest of the tutorial. *Since some menus are visited several times, please follow the directions rather than explore the system.*

At this point, the subjects filled out the Procedure Representation Worksheet.

Continue with the tutorial when you have finished the above worksheet. *Remember to follow the directions instead of exploring the system.*

When you have finished the tutorial, stop. Indicate that you are done and ask to continue.

Please look over the tutorial's synopsis.

Do you have any questions?

End of the first day

Second Day Directions

Please review the tutorial's synopsis (chapter 9) and the worksheet on procedure representation. You should focus on those two sections but you can look at other parts of the tutorial. Do not spend more than ten minutes. **Stop when you are finished. Indicate that you are done and ask to continue.**

Now go over the second day tutorial. **Stop when you are finished. Indicate that you are done and ask to continue.**

At the end of the second day tutorial, subjects solve the practice problem (section B.11 and look at its solution (section B.12).

Do you have any questions?

Authoring

Now you will author two procedures. For each procedure, you should

1. Enter the procedure and modify it until you're satisfied.

2. Test it.

3. Indicate when you are finished with it.

You cannot spend more than 30 minutes on a procedure.

Instructions

These are the directions given to the subjects who both demonstrated and experimented. If a subject did not demonstrate or experiment, then directions that mention demonstrations or experiments were removed.

Remember to consult the tutorial's synopsis chapter if you have questions.

Please do not change status values between "provisional" and "required." Both values indicate that the object will be used.

When authoring, remember that we are primarily concerned with attributes that change value during the procedure.

A procedure should only contain necessary ordering relationships.

When demonstrating a procedure, make sure that a step has been processed before demonstrating the next step. This can be done by making sure that the text "wait2" and "wait3" is scrolling in the Soar window. *Demonstrating the next step too quickly can cause serious problems.* A good rule of thumb is at least 5 to 10 seconds between steps.

Also remember to let Diligent "experiment" with the procedure. After experimenting, you need to derive the ordering relationships so that they reflect what was learned during the experiments.

Because your activities are being monitored, focus on authoring procedures rather than exploring the system out of curiosity.

Assume that each procedure starts in the state shown in the Vista window. The procedure's description assumes that you start in that state.

Please give each procedure a distinct name.

You will now be given

- A description of the procedures to be authored.

- Pictures of the device with labels identifying the names of various objects.

- A description of all attributes and their legal values.

Stop and indicate that you are ready to continue. The person helping you will prepare the system for the first procedure.

Now author the "High Condensate Level Shutdown" procedure.

Stop when you have finished with the procedure. Indicate that you are done and ask to continue. The person helping you will prepare the system for the second procedure.

Now author the "Overload Relay Tripped" procedure.

Stop when you have finished with the procedure. Indicate that you are done and ask to continue. The person helping you will save the second procedure.

Go to the next page and fill out the questionnaire.

At this point the subjects, filled out the post-test.

B.7 The List of Attribute Values

This list of attribute values was was given to all subjects, but was only required by subjects who only used an editor. The list provides some indication of the size and complexity of the domain.

```
# The following list provides descriptions and values for the HPAC
# attributes.

  Attribute Name      Description
Values
--------------------------------------------------------------


cdm_chnl1_lt_state     "first stage alarm light"
"on"
"off"
cdm_chnl2_lt_state     "second stage alarm light"
"on"
"off"
cdm_chnl3_lt_state     "third stage alarm light"
"on"
"off"
cdm_chnl4_lt_state     "fourth stage alarm light"
"on"
"off"
cdm_power_state        "condensate drain monitor power"
"on"
"off"
cdm_status             "condensate drain monitor status"
"system reset"
"function test"
"halted"
cp_oil_level           "oil level"
"normal"
"low"
"high"
ctrl_mon_sel_state    "compressor mode"
"monitored"
"unmonitored"
ctrl_motor_status     "motor"
"on"
"off"
ctrl_power_status "power"
"on"
"off"
ctrl_relayreset_state "overload relay"
```

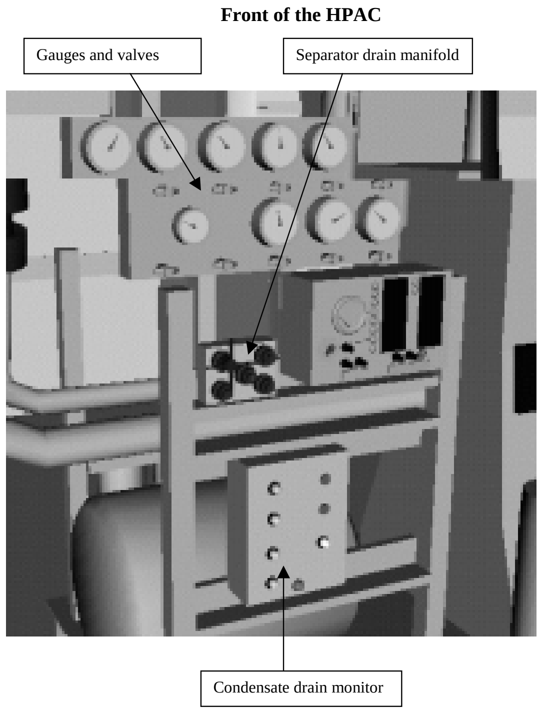
```
"ok"
"tripped"
dipstick_position      "dipstick position"
"in"
"halfway"
"out"
gb_air1_state       "first air intake valve"
"open"
"shut"
gb_air2_state       "second air intake valve"
"open"
"shut"
gb_covstg1_state       "first cutout valve"
"open"
"shut"
gb_covstg2_state       "second cutout valve"
"open"
"shut"
gb_covstg3_state       "third cutout valve"
"open"
"shut"
gb_covstg4_state       "fourth cutout valve"
"open"
"shut"
gb_covstg5_state       "fifth cutout valve"
"open"
"shut"
sdm_handle_location    "location of the handle"
"separator drain 1st stage valve"
"separator drain 2nd stage valve"
"separator drain 3rd stage valve"
"separator drain 4th stage valve"
"separator drain 5th stage valve"
sdm_sep_drnvlv1_pressure  "first stage pressure"
"high"
"normal"
sdm_sep_drnvlv1_state  "first stage valve"
"open"
"shut"
sdm_sep_drnvlv2_pressure  "second stage pressure"
"high"
"normal"
sdm_sep_drnvlv2_state  "second stage valve"
"open"
"shut"
```

```
sdm_sep_drnvlv3_pressure  "third stage pressure"
"high"
"normal"
sdm_sep_drnvlv3_state  "third stage valve"
"open"
"shut"
sdm_sep_drnvlv4_pressure  "fourth stage pressure"
"high"
"normal"
sdm_sep_drnvlv4_state  "fourth stage valve"
"open"
"shut"
sdm_sep_drnvlv5_state  "fifth stage valve"
"open"
"shut"
student_speaking       "student speaking"
"true"
"false"
surge_tank_level       "surge tank level"
"empty"
"normal"
"full"
tm_ltcrkcsoil_state    "indicator light"
"on"
"off"
tm_ltdis1_state        "indicator light"
"on"
"off"
tm_ltdis2_state        "indicator light"
"on"
"off"
tm_ltdis3_state        "indicator light"
"on"
"off"
tm_ltdis4_state        "indicator light"
"on"
"off"
tm_ltdis5_state        "indicator light"
"on"
"off"
tm_ltfindis_state      "indicator light"
"on"
"off"
tm_ltjkwtrout_state    "indicator light"
"on"
```

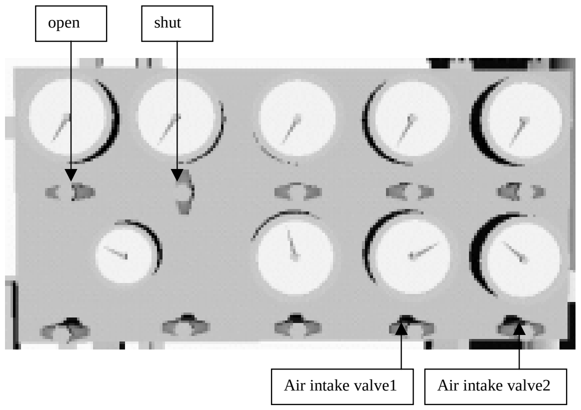
```
"off"
tm_ltsuc1_state        "indicator light"
"on"
"off"
tm_ltsuc2_state        "indicator light"
"on"
"off"
tm_power_state         "temperature monitor power"
"on"
"off"
tm_status              "temperature monitor status"
"test 100"
"test 350"
NONE
"reset"
"testing"
"test trip temperature"
```

B.8 Labeled Pictures of the HPAC

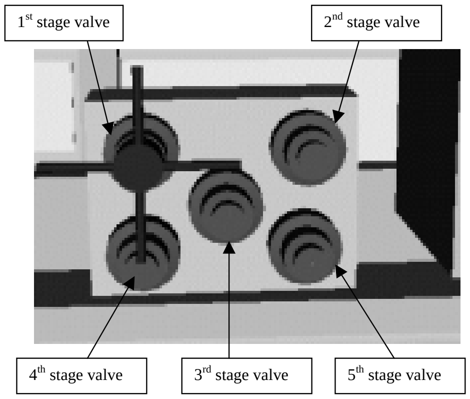
The following pictures were given to test subjects.

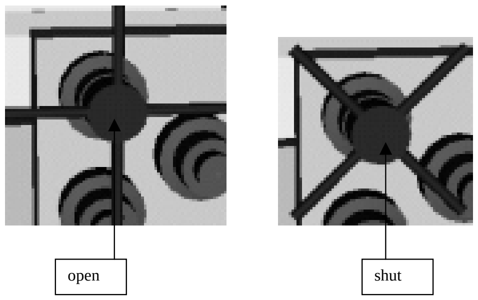


Gauges and Valves

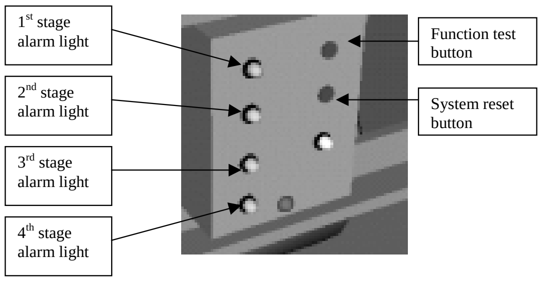


Separator Drain Manifold

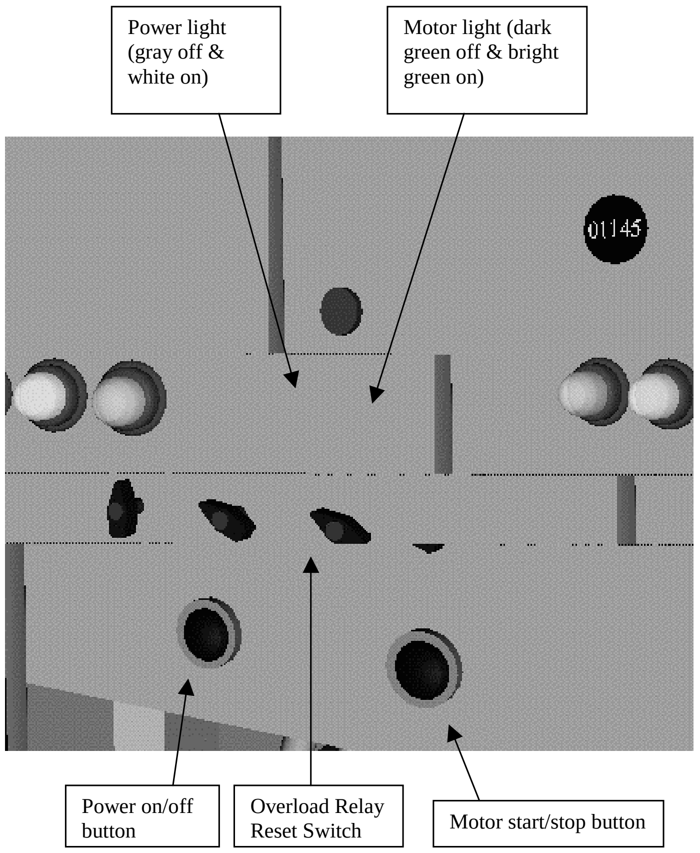




Condensate Drain Monitor



Control Door


Power light (gray off & white on)
Motor light (dark green off & bright green on)
01145
Power on/off button
Overload Relay Reset Switch
Motor start/stop button

Latex is not formatting the picture properly.

B.9 Procedure Descriptions

This section contains the procedure descriptions that were given to test subjects. The first procedure authored is High Condensate Level Shutdown, and second procedure authored is Overload Relay Tripped.

B.9.1 High Condensate Level Shutdown

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. At this point, some alarm lights on the drain monitor's panel turn red.

The alarm lights will turn off only after the pressure is relieved. For each alarm light that is red, the student can relieve the pressure by opening the separator drain manifold valve that corresponds to that alarm light. Once the pressure is relieved, valves should be shut for normal operations.

Once the motor has been started with the control door panel's motor button, the pressure will be relieved and the alarm lights will turn off. Before starting the motor, the student should reset the drain monitor by pressing the drain monitor panel's system reset button.

The procedure's initial state can be seen in the Vista window. Initially, high levels of condensation have caused the motor to turn off and two alarm lights to turn red.

When performing the procedure, the student will need to both open valves and turn on the motor.

When you are finished, the alarm lights should be off, the valves should be shut, and the motor should be running.

Reminder: you will only be asked to name an operator the first time the operator's action is used. In other words, if the action is used again, you will not be asked for an operator name.

B.9.2 Overload Relay Tripped

When the compressor gets overloaded, a relay will trip and turn off the motor.

At this point, the compressor's electronics may be in an anomalous state. The student can correct state by turning off the power with the power button on the control door panel. The button is a toggle that turns the power on or off.

One reason for overload is too much air pressure. To limit the air pressure, the student should shut the two air intake valves.

Once the relay is tripped, the compressor will not work until the relay switch on the control door panel is toggled. In order to make sure that the power has been turned off, the relay will not reset unless the power is off.

Once the relay has been reset, the student should turn the power on and then start the motor with the control door panel's motor button.

When you are finished, the air intake valves should be shut and the motor should be running.

B.10 Desired Procedures

This section contains the desired procedures against which the subjects are evaluated.

B.10.1 High Condensate Level Shutdown

The first procedure restarts the motor after high condensate pressure has shut it done. The desired procedure has the following steps:

1. Turn the handle and open the second stage valve. Do this by selecting the handle.
2. Move the handle to the first stage valve. Do this by selecting the first stage valve.
3. Turn the handle and open the first stage valve.
4. Press the reset button.
5. Press the motor button and turn the motor on.
6. Turn the handle so that first stage valve is shut.
7. Move the handle to the second stage valve.
8. Turn the handle so that the second stage valve is shut.

The plan for the procedure is as follows:

Steps:
begin-clsd turn-1 move-1st-2 turn-3 reset-4 motor-5
turn-6 move-2nd-7 turn-8 end-clsd

Ordering Constraints:

1.	turn-1	before	move-1st-2
2.	turn-1	before	motor-5
3.	turn-1	before	turn-8
4.	move-1st-2	before	turn-3
5.	move-1st-2	before	turn-6
6.	turn-3	before	motor-5
7.	turn-3	before	turn-6
8.	turn-3	before	move-2nd-7
9.	reset-4	before	motor-5
10.	motor-5	before	turn-6
11.	motor-5	before	turn-8
12.	turn-6	before	move-2nd-7
13.	move-2nd-7	before	turn-8

Causal Links:

1. begin-clsd establishes (cdm_chnl2_lt_state on) for turn-1
2. begin-clsd establishes (sdm_handle_location “separator drain 2nd stage valve”) for turn-1
3. begin-clsd establishes (sdm_sep_drnvlv2_state shut) for turn-1
4. begin-clsd establishes (cdm_chnl1_lt_state on) for turn-3
5. begin-clsd establishes (sdm_sep_drnvlv1_state shut) for turn-3
6. begin-clsd establishes (cdm_status halted) for reset-4
7. begin-clsd establishes (cdm_chnl1_lt_state on) for motor-5
8. begin-clsd establishes (cdm_chnl2_lt_state on) for motor-5
9. begin-clsd establishes ((ctrl_motor_status off) for motor-5
10. begin-clsd establishes (sdm_sep_drnvlv1_pressure high) for motor-5
11. begin-clsd establishes (sdm_sep_drnvlv2_pressure high) for motor-5
12. turn-1 establishes (sdm_sep_drnvlv2_state open) for motor-5
13. turn-1 establishes (sdm_sep_drnvlv2_state open) for turn-8
14. move-1st-2 establishes (sdm_handle_location “separator drain 1st stage valve”) for turn-3
15. move-1st-2 establishes (sdm_handle_location “separator drain 1st stage valve”) for turn-6
16. turn-3 establishes (sdm_sep_drnvlv1_state open) for motor-5
17. turn-3 establishes (sdm_sep_drnvlv1_state open) for turn-6
18. reset-4 establishes (cdm_status “system reset) for motor-5
19. reset-4 establishes (cdm_status “system reset) for end-clsd

20. motor-5	establishes (cdm_chnl1_lt_state off) for turn-6
21. motor-5	establishes (cdm_chnl2_lt_state off) for turn-8
22. motor-5	establishes (cdm_chnl1_lt_state off) for end-clsd
23. motor-5	establishes (cdm_chnl2_lt_state off) for end-clsd
24. motor-5	establishes (ctrl_motor_status on) for end-clsd
25. motor-5	establishes (sdm_sep_drnvlv1_pressure normal) for end-clsd
26. motor-5	establishes (sdm_sep_drnvlv2_pressure normal) for end-clsd
27. turn-6	establishes (sdm_sep_drnvlv1_state shut) for end-clsd
28. move-2nd-7	establishes (sdm_handle_location "separator drain 2nd stage valve") for turn-8
29. move-2nd-7	establishes (sdm_handle_location "separator drain 2nd stage valve") for end-clsd
30. turn-8	establishes (sdm_sep_drnvlv2_state shut) for end-clsd

B.10.2 Overload Relay Tripped

The second procedure restarts the motor after high air pressure has caused relay to trip. The desired procedure has the following steps:

1. Shut the first air intake valve.
2. Shut the second air intake valve.
3. Turn off the power.
4. Toggle the relay reset switch.
5. Turn on the power.
6. Turn on the motor.

The plan for the procedure is as follows:

Steps:
begin-rlytp air1-1 air-2 power-3 reset-4 power-5
motor-6 end-rlytp

Ordering Constraints:

1.	air1-1	before	motor-6
2.	air2-2	before	motor-6
3.	power-3	before	power-5
4.	power-3	before	reset-4
5.	reset-4	before	power-5
6.	reset-4	before	motor-6
7.	power-5	before	motor-6

Causal Links:

1.	begin-rlytp	establishes (gb_air1_state open)	for air1-1
2.	begin-rlytp	establishes (gb_air2_state open)	for air2-2
3.	begin-rlytp	establishes (ctrl_power_status on)	for power-3
4.	begin-rlytp	establishes (ctrl_relayreset_status tripped)	for reset-4
5.	begin-rlytp	establishes (ctrl_motor_status off)	for motor-6
6.	air1-1	establishes (gb_air1_state shut)	for motor-6
7.	air1-1	establishes (gb_air1_state shut)	for end-rlytp
8.	air2-2	establishes (gb_air2_state shut)	for motor-6
9.	air2-2	establishes (gb_air2_state shut)	for end-rlytp
10.	power-3	establishes (ctrl_power_status off)	for reset-4
11.	power-3	establishes (ctrl_power_status off)	for power-5
12.	reset-4	establishes (ctrl_relayreset_status ok)	for motor-6
13.	reset-4	establishes (ctrl_relayreset_status ok)	for end-rlytp
14.	power-5	establishes (ctrl_power_status on)	for motor-6
15.	power-5	establishes (ctrl_power_status on)	for end-rlytp
16.	motor-6	establishes (ctrl_motor_status on)	for end-rlytp

B.11 Practice Procedure

At the end of the second day's training, subjects solved the following problem.

Up to this point you have been following very specific instructions. In this section you are going to author with only general directions. To author the procedure, you need define and test it.

Like the above procedure, the practice procedure will toggle two cutout valves. However, instead of toggling the first and second cutout valves, you will now toggle the third and fourth cutout valves, which are to the right of the second cutout valve. The attributes "gb_covstg3_state" and "gb_covstg4_state" should initially be "open" and should be "shut" when the procedure is finished.

You have 10 minutes to finish this task.

B.12 Practice Procedure Solution

Looking at the practice problem's solution was last thing that subject's did during training. Only the subject's who used demonstrations or experiments saw the parts of the solution that mention demonstrations or experiments.

The following is a solution for the practice procedure. You should compare your procedure against the solution. Ask questions if you do not understand why this is a reasonable solution.

Let the procedure be called "practice

Steps: (execution order)

(The order that steps are toggled doesn't matter.)

begin-practice, toggle-3rd-3, toggle-4th-4, end-practice

Causal links:

begin-practice establishes "gb_covstg3_state = open" for toggle-3rd-3

begin-practice establishes "gb_covstg4_state = open" for toggle-4th-4

toggle-3rd-3 establishes "gb_covstg3_state = shut" for end-practice

toggle-4th-4 establishes "gb_covstg4_state = shut" for end-practice

Ordering Constraints:

None. (All are "ignored" because they involve the steps begin-practice and end-practice.)

Operators:

toggle-3rd

Preconditions: (gb_covstg3_state = open)

State changes: (gb_covstg3_state = shut)

toggle-4th

Preconditions: (gb_covstg4_state = open)

State changes: (gb_covstg4_state = shut)

Step Prerequisite preconditions:

None are necessary.

Number of demonstrations:

Only one is necessary.

Continued on next page.

Experiments resulted in:

The removal of one incorrect precondition, one incorrect causal link, and one incorrect ordering constraint.

Appendix C

Evaluation Data

The following contains data for the three experimental conditions. Experimental condition EC_1 allows demonstrations and experiments. Condition EC_2 allows demonstrations but not experiments. Condition EC_3 uses only an editor.

In order to protect the privacy of subjects, the masculine pronoun "he" will always be used when referring to a subject. The use of "he" does not indicate whether the subject was male or female.

C.1 Background Questionnaire

This data has been withheld to protect the privacy of the subjects. The data is summarized in section 7.5.1.

C.2 Impressions of Diligent

This section contains the data describing the subjects' impressions of Diligent. The last activity that subjects performed were answering these questions, which are located at the end of the subjects' directions (appendix B).

Because some of the questions are inappropriate for some of the experiment conditions, the subjects in each experimental condition answered a different subset of questions.

The answers listed in the following tables represent the following questions. An answer of 1 means not at all, 4 means somewhat, and 7 means a great deal.

- I1 ≡ General Questions about Authoring
  - I1a ≡ Like the system
  - I1b ≡ Easy to use
  - I1c ≡ Easy to specify a step
  - I1d ≡ Easy to identify a step's preconditions
  - I1e ≡ Easy to identify a step's state changes
  - I1f ≡ Easy to identify how operators influenced a step's preconditions and state changes
- I2 ≡ Questions about demonstrating
  - I2a ≡ Easy to demonstrate
  - I2b ≡ Were additional demonstrations useful
- I3 ≡ Questions about experiments
  - I3a ≡ Did you like experimenting
  - I3b ≡ Where experiments quick enough
  - I3c ≡ Did experiments save work
  - I3d ≡ Did experiments find errors that would have been missed

Item I3b is different than what the questionnaire asked. The questionnaire asked, "Did experiments take too long." So that the data is easier to interpret, question was reformulated so that a lower answer is less positive. When transforming the answers from the questionnaire to I3b, the following mappings were used: 1 to 7, 2 to 6, 3 to 5 and 4 to 4.

C.2.1 Experimental Condition EC_1

subject	I1a	I1b	I1c	I1d	I1e	I1f
3	6	2	3	2	4	4
6	5	3	4	5	2	4
12	4	4	5	4	7	6
14	6	6	7	6	6	6

Table C.1: EC_1 Impressions about Authoring

subject	I2a	I2b	I3a	I3b	I3c	I3d
3	2	5	4	6	7	1
6	4	N/A	5	2	N/A	N/A
12	7	7	1	7	4	4
14	7	1	7	7	7	4

Table C.2: EC_1 Impressions about Demonstrations and Experiments

Subject 3 didn't feel that error recovery was covered well enough in training. The subject also felt that the user interface was confusing.

Subject 6 had difficulty demonstrating while remembering the initial state. The subject also felt that the ambiguous use of terms made things more difficult.

Subject 12 answered the questions about experiments (i.e. I3a-I3d) with yes or no rather than a number. In the table, 1 is used for no, and 4 is used for yes. Because the data for I3b was transformed, the subject's "no" became a 7.

Subject 14 answered question I3d with yes rather than a number. In the table 4 is used for yes.

Subject 14 didn't understand how helpful experiments can be during training. Instead, the subject learned how helpful experiments can be during the evaluation's first procedure.

C.2.2 Experimental Condition EC_2

subject	I1a	I1b	I1c	I1d	I1e	I1f
2	4	4	7	7	5	5
5	5	2	4	5	5	3
8	3	4	1	3	4	3
9	4	5	7	3	3	3
10	3	3	6	6	6	5
11	5	2	4	5	5	3

Table C.3: EC_2 Impressions about Authoring

subject	I2a	I2b	I3a	I3b	I3c	I3d
2	4	4	-	-	-	-
5	6	6	-	-	-	-
8	1	4	-	-	-	-
9	4	5	-	-	-	-
10	7	3	-	-	-	-
11	3	5	4	2	N/A	4

Table C.4: EC_2 Impressions about Demonstrations and Experiments

Subject 2 complained that the environment was slow to react.

Subject 5 had a few complaints. The descriptions of the procedures authored during the experiment were "somewhat unclear." The environment was unresponsive; the subject felt that manipulating an object required the mouse to be clicked in small region. The subject also had to be told whether lights in the environment were turned on.

Subject 8 had difficulty correcting mistakes. The subject felt that there were a lot of distractions. The subject also felt that the system was slow and unresponsive. (This comment appears to be directed at the environment.) However, the subject liked the GUI and wrote that the GUI, "allowed a feel of ease of use that didn't always come across in [training]."

Subject 9 would have really preferred to use an editor (e.g. EC_3) instead of demonstrating procedures. The subject wanted to specify all the steps before dealing preconditions and state changes. The subject wrote, "The system seems to have several features to automatically do several things, but they are not very useful. I would have liked to specify my initial state and final state option and then go on to define my steps, so I need not be concerned with preconditions."

Subject 10 couldn't figure out how to make a step optional so that it would be skipped if it wasn't needed. This a misunderstanding of procedural presentation. The subject had difficulty with the experiment's procedure descriptions; it was "not easy" to determine the steps or the minimal dependencies between the steps. The subject also had problems removing extraneous dependencies because the removed dependencies didn't immediately disappear. (The comment on removed dependencies may be related to the fact that the window containing a procedure's graphical representation does not update its graph. To update the graph, a subject needs to close and re-opening the window.)

Subject 11 was in EC_1, but never used experiments. For this reason, the subject was moved to EC_2. The subject said that he didn't know that experiments would remove excess causal links.

C.2.3 Experimental Condition EC_3

subject	I1a	I1b	I1c	I1d	I1e	I1f
1	4	2	2	2	2	2
4	5	3	5	6	6	5
7	3	3	5	6	6	3
13	2	2	4	4	4	1
16	6	4	6	6	6	3

Table C.5: EC_3 Impressions about Authoring

Subject 1 was confused in a number of areas. The subject was didn't understand the operator's and steps. The subject didn't understand why both were needed and whether the relationships was one-to-one or many-to-one. (This is the only subject that did not fill out the procedural representation worksheet during the first day's training.) The subject was also confused on how to insert a step in front of another step.

Subject 5 would have liked to use templates for steps with similar preconditions and state changes. The subject wrote, "The testing and explanation components were very good."

Subject 7 had quite a few problems. The subject had difficulty familiarizing himself with the domain's attribute names. The similarity of these names made things more difficult. The subject wrote, "Having predetermined names for the actions actually disoriented me when solving the the problems." The subject also had problems with the environment. It was difficult to zoom in or out. It was also difficult to determine where to click the mouse when manipulating an object.

Subject 13 had a number of user interface problems. The subject wanted a button to press for help. The subject was frustrated because he couldn't figure out how to delete unwanted conditional effects (you can't). The subject felt that step preconditions could only be added when the procedure is graphically displayed. (This probably reflects the fact that the procedure's graph is not updated until the graph's window is closed and re-opened.) The subject was also irritated that windows didn't open more quickly.

Subject 16 had problems with ambiguously ordered steps. In the second procedure, the subject felt that he had to order the steps "illogically." However, the subject felt that system was "easy to use once understood." The subject also felt that the second procedure was easier because the first procedure involved a "steep learning curve" on "parts of the environment and the linking of more complex sets of steps."

C.3 Authoring

This section contains the data describing how the subjects authored during the experiment.

The answers listed in the following tables represent the following data. Except for some of the time values, each procedure of the two procedures has the following data.

Sometimes a subject would abandon a flawed procedure and create a new procedure. When this happens, the edits for the abandoned procedure are still counted.

Edits:
- ed1 ≡ Steps added in normal demonstration
- ed2 ≡ Steps added in clarification demonstration (EC_1 and EC_2 only)
- ed3 ≡ Actions in prefix before start of demonstration (EC_1 and EC_2 only)
- ed4 ≡ Deleted steps
- ed5 ≡ Edits to causal links
- ed6 ≡ Edits to ordering constraints
- ed7 ≡ Edits to goal conditions
- ed8 ≡ Edits to filter attributes out of causal links
- ed9 ≡ Edits to filter attributes out of ordering constraints
- ed10 ≡ Edits to conditional effect preconditions
- ed11 ≡ Edits to conditional effect state changes (EC_3 only)
- ed12 ≡ Edits to control preconditions (associated with steps rather than conditional effects)
- ed13 ≡ Edits to associate conditional effects to steps (EC_3 only)
- ed14 ≡ Total logical edits. This is the sum of ed1 – ed12. ed13 is ignored out of concerns for fairness.

Experiments: (EC_1 only)
- exp1 ≡ Prefix actions performed preparing experiments
- exp1 ≡ Steps performed during experiments experiments

Errors:
- er1 missing ordering constraints
- er2 unnecessary or incorrect ordering constraints
- er3 missing causal links
- er4 unnecessary or incorrect causal links
- er5 missing steps
- er6 unnecessary or incorrect steps
- er7 total errors of omission (i.e. missing objects: er1 + er3 + er5)
- er8 total errors of commission (i.e. unnecessary or incorrect objects: er2 + er4 + er6)
- er9 total errors (er7 + er8)

a1 ≡ Number of steps in the procedure

a2 ≡ Could the final procedure be demonstrated a3 $\equiv$ total effort (total edits(ed14) + total errors(er9))

Time: (in minutes)

- t1 $\equiv$ First day training time
- t2 $\equiv$ Second day training time
- t3 $\equiv$ Total training time
- t4 $\equiv$ 1st procedure time before testing
- t5 $\equiv$ 1nd procedure total time
- t6 $\equiv$ 2nd procedure time before testing
- t7 $\equiv$ 2nd procedure total time

In the following tables, the authoring data represents two times: when testing starts and when the procedure is finished. If only one value is given, then both are the same. When a value is of the form A/B, the two values are different. The value at the start of testing is A, and the value at the end is B.

The times are derived from both log files and notes taken while the subject was training and authoring. Some of the times may be off by $\pm$ two minutes. The error is this large because some of the times had to be explicitly logged and because some of the times came from the notes. When a time was logged, sometimes the procedure had to put into the proper state, which involved closing windows and deriving the procedure's goals and causal links from the current database.

Times that were explicitly logged include starting training, finishing training, starting a procedure and ending a procedure. However, the total time allowed for authoring a procedure was measured with an alarm clock.

The start of the training time for the first session is when the subject sits down. This means that first session's training time includes the 5 to 10 minutes required to fill out the background questionnaire.

C.3.1 Experimental Condition EC_1

Topic	**Subject**			
	3	**6**	**12**	**14**
ed1	11	5	8	8
ed2	0	0	0	0
ed3	4	0	0	0
ed4	0	2	0	0
ed5	0	0	1	0
ed6	0	0	0	0
ed7	0	0	1	0
ed8	0	0	0	0
ed9	0	0	0	0
ed10	0	0	0	0
ed11	-	-	-	-
ed12	0	5	1	0
ed13	-	-	-	-
ed14	15	12	11	8
exp1	0	0	0	0
exp2	25	0	49	49
er1	10	12	0	0
er2	3	2	3	3
er3	24	28	5	4
er4	6	7	4	3
er5	3	6	0	0
er6	2	1	0	0
er7	37	46	5	4
er8	11	10	7	6
er9	48	56	12	10
a1	7	3	8	8
a2	no	no	yes	yes
a3	63	68	23	18

Table C.6: EC_1 Procedure 1 Authoring Information

Topic	**Subject**			
	3	**6**	**12**	**14**
ed1	5	4	5	6
ed2	0	0	0	0
ed3	0	0	0	0
ed4	0	0	0	0
ed5	0	0	0	0
ed6	0	1	0	0
ed7	1	1	0	0
ed8	0	0	0	0
ed9	0	0	0	0
ed10	0	0	0	0
ed11	-	-	-	-
ed12	0	3/4	4	5
ed13	-	-	-	-
ed14	6	9/10	9	11
exp1	0	0	0	0
exp2	16	9	16	25
er1	6	5	4	0
er2	0	0	4	4
er3	8	13	6	0
er4	0	1/2	2	4
er5	1	2	1	0
er6	0	0	0	0
er7	15	20	11	0
er8	0	1/2	6	8
er9	15	21/22	17	8
a1	5	4	5	6
a2	no	no	no	yes
a3	21	30/32	26	19

Table C.7: EC_1 Procedure 2 Authoring Information

Topic	**Subject**			
	3	**6**	**12**	**14**
t1	93	100	149	120
t2	40	48	34	53
t3	133	148	183	173
t4	28	30	30	21
t5	30	30	30	25
t6	22	26	18	20
t7	29	30	18	22

Table C.8: EC_1 Time Spent on Activities

C.3.2 Experimental Condition EC_2

Topic	Subject					
	2	**5**	**8**	**9**	**10**	**11**
ed1	17	-	-	-	29	10
ed2	0	-	-	-	0	0
ed3	0	-	-	-	1	0
ed4	0	-	-	-	0	0
ed5	0	-	-	-	0	0
ed6	0	-	-	-	4	14
ed7	0	-	-	-	2	0
ed8	0	-	-	-	0	0
ed9	0	-	-	-	0	0
ed10	0	-	-	-	8	0
ed11	-	-	-	-	-	-
ed12	0	-	-	-	0	2
ed13	-	-	-	-	-	-
ed14	17	-	-	-	44	26
exp1	-	-	-	-	-	-
exp2	-	-	-	-	-	-
er1	8	-	-	-	4	3
er2	18	-	-	-	7	12
er3	11	-	-	-	8	3
er4	26	-	-	-	5	25
er5	3	-	-	-	0	0
er6	5	-	-	-	1	2
er7	22	-	-	-	12	6
er8	49	-	-	-	13	39
er9	71	-	-	-	25	45
a1	10	-	19	-	9	10
a2	no	-	yes	no	yes	yes
a3	88	-	-	-	69	71

Table C.9: EC_2 Procedure 1 Authoring Information

Subject 5 demonstrated the steps too quickly. Diligent's implementation could not determine which state changes were caused by a given action. To correct this, the subject would have had to empty Diligent's knowledge base and start over.

Subject 8 didn't understand the directions. The subject tried to move the valve handle to every valve. The procedure is so bad that it cannot be easily graded. The procedure had 51 edits and at least 65 errors.

Subject 9 authored a hierarchical procedure with several subprocedures. This makes it difficult to compare the procedure to the other subjects, who did not attempt a hierarchical procedure. The procedure has two problems: 1) the subject performed unnecessary steps that moved the handle between the valves, and 2) the subject forgot to turn on the motor at the end of the procedure.

Topic	**Subject**					
	2	**5**	**8**	**9**	**10**	**11**
ed1	7	11	6	7	6	6
ed2	0	0/6	2	0	0	0
ed3	0	0	0	0	0	0
ed4	0	0/5	0	0	0	0
ed5	0	0	0	0	1	0/1
ed6	0	0	10	4	2	0/3
ed7	0	1	0	0	2	0
ed8	0	0	0	0	0	0
ed9	0	0	0	0	0	0
ed10	0	0	0	0	3/12	0
ed11	-	-	-	-	-	-
ed12	0	0	2	1	0	0/6
ed13	-	-	-	-	-	-
ed14	7	12/23	20	12	14/23	6/16
exp1	-	-	-	-	-	-
exp2	-	-	-	-	-	-
er1	0	0	3	3	0/7	0/1
er2	7	7	3	4	4/0	7/6
er3	0	1/0	7	5	2/10	0
er4	8	8	4	3	5/0	8/7
er5	0	0	0	0	0	0
er6	1	5/0	0	0	0	0
er7	0	1/0	10	8	2/17	0/1
er8	16	20/15	7	7	9/0	15/13
er9	16	21/15	17	15	11/17	15/14
a1	7	11/6	6	6	6	6
a2	yes	yes	no	yes	yes	yes
a3	23	33/38	37	27	25/40	21/30

Table C.10: EC_2 Procedure 2 Authoring Information

The final procedure for subject 10 is marked as working because the steps are in the correct order and all ordering constraints are reasonable. However, the final procedure is basically unordered. (The version before testing was much better.)

Topic	**Subject**					
	2	**5**	**8**	**9**	**10**	**11**
t1	119	75	84	91	91	123
t2	38	27	35	66	39	54
t3	157	102	119	157	130	177
t4	24	30	30	30	30	30
t5	29	30	30	30	30	30
t6	5	15	30	30	17	19
t7	8	30	30	30	30	29

Table C.11: EC_2 Time Spent on Activities

Subject 9 skipped a day and took longer to train on the second day because of software problems.

C.3.3 Experimental Condition EC_3

Topic	Subject				
	1	**4**	**7**	**13**	**16**
ed1	3	4	10	11	7
ed2	-	-	-	-	-
ed3	-	-	-	-	-
ed4	0	0	6	4	3
ed5	0	0	0	0	0
ed6	0	0	0	0	0
ed7	0	0	0	0	0
ed8	0	0	0	0	0
ed9	0	0	0	0	0
ed10	3	24	0	21	11
ed11	1	12	2	7	11
ed12	0	0	0	0	0
ed13	3	12	6	11	11
ed14	7	40	18	43	32
exp1	-	-	-	-	-
exp2	-	-	-	-	-
er1	13	13	13	13	13
er2	0	1	0	0	2
er3	30	30	29	29	27
er4	1	5	0	7	6
er5	7	6	5	3	4
er6	2	2	0	2	0
er7	50	49	47	45	44
er8	3	8	0	9	8
er9	53	57	47	54	52
a1	3	4	3	7	4
a2	no	no	no	no	no
a3	60	97	63	97	84

Table C.12: EC_3 Procedure 1 Authoring Information

The correct procedure has 8 steps.

Subject 1 didn't have any ed13 data so the value of ed13 equals the number of steps.

For subject 7, it is not clear why (ed1 - ed4 = 4) rather than 3 (a1). This discrepancy was not reproducible. Subject 7 did so well because he didn't do much authoring. For example, the subject did not specify a single precondition (ed10 and ed12).

Topic	**Subject**				
	1	**4**	**7**	**13**	**16**
ed1	7	8	6	6	6
ed2	-	-	-	-	-
ed3	-	-	-	-	-
ed4	1	0	0	0	0
ed5	0	0	0	0	0
ed6	0	0	0	0	0
ed7	0	0	0	0	1/2
ed8	0	0	0	0	0
ed9	0	0	0	0	0
ed10	9	16	8	8	6
ed11	8	8	9	6	6
ed12	0	0	1	0/2	3/7
ed13	6	11	7	6	6
ed14	25	32	24	20/22	22/27
exp1	-	-	-	-	-
exp2	-	-	-	-	-
er1	7	5	6	3	2
er2	0	6	0	0/4	0
er3	11	8	7	3	5/6
er4	3	12	2	0/2	0
er5	0	0	1	0	0
er6	0	2	0	0	0
er7	18	13	14	6	7/8
er8	3	20	2	0/6	0
er9	21	33	16	6/12	7/8
a1	6	8	5	6	6
a2	no	no	no	yes	yes
a3	46	65	40	26/34	29/35

Table C.13: EC_3 Procedure 2 Authoring Information

The correct procedure has 6 steps.

Subject 1 didn't have ed13 data so the value of ed13 equals the number of steps.

For subject 7, it is not clear why (ed1 - ed4 = 6) rather than 5 (a1). This discrepancy was not reproducible.

Topic	**Subject**				
	1	**4**	**7**	**13**	**16**
t1	117	114	117	67	72
t2	46	47	43	22	35
t3	163	161	160	89	107
t4	30	30	30	28	30
t5	30	30	30	28	30
t6	30	30	30	14	19
t7	30	30	30	19	30

Table C.14: EC_3 Time Spent on Activities

C.4 Session Log

This section contains data collected during each subject's two sessions. The section also mentions changes to the system and training to correct problems with earlier subjects.

The changes were meant to correct problems with the study. First, it was important that subjects understood how to correctly use Diligent. Second, subjects needed to understand what steps were needed in the two procedures being authored.

Two changes that dealing with how subjects authored are not mentioned. One change is repeatedly reminding EC_1 and EC_2 subjects to avoid demonstrating too quickly. Demonstrating too quickly caused problems with Diligent's implementation. In particular, it caused pairs of actions to appear simultaneous, and Diligent does not handle simultaneous actions. The other change is telling EC_1 subjects to experiment with their procedures. One EC_1 subject, who didn't experiment, was switched group EC_2.

The potential for simultaneous actions was aggravated by a memory leak involving the VIVIDS simulation and the Vista browser. As more memory was lost, the Vista would get progressively slower and less responsive. Shortly after subject 7, updated versions of VIVIDS and Vista were installed. This fixed many of the performance problems that subjects experienced with Vista.

The material in this section is derived from notes rather than the answers to the questionnaire on the subject's impressions of Diligent. In the following, the experimenter/author is referred to as the *test monitor*. Minor errors in manuals, such as typographical and grammatical errors, are not mentioned.

- *Subject 1.*
  - Session 1

    The subject had questions about using Vista (the environment's graphical interface).

    The subject looked at menus that hadn't been discussed yet. The test monitor told the subject, "it will become clear later on."

    The subject was confused that the graph of the procedure was not updated when a step was added. (The graph is not updated after the window is opened.)

    The subject had difficulty understanding the concepts involved in a authoring procedure. Part of the reason is that he didn't know what he was trying to produce. He also had difficulty connecting a graph of a procedure with STEVE's explanation.

    The subject felt that he was having to simultaneously learn the procedural representation and how to use Diligent. The subject felt that he could do this, but other subjects might have more problems. (This comment caused the creation of the procedural representation section and worksheet.)
  - Session 2: training

    The subject read the procedural representation section (which later subjects read during the first session).

    Problems zooming in and out in Vista.

    During the practice problem, the subject was told to test his procedure.

  - Session 2: 1st procedure

    Confused about which steps to perform and their order.

    Expressed a desire for a list of available actions. (No subject was given this list. For this group (EC_3), the available actions are listed in one of Diligent's menus.)

- *Changes*

  The procedural representation section and worksheet were added to the first day's tutorial.

- *Subject 2*

  - *Session 1*

    The subject was confused about how he could tell whether a precondition is correct or not. The training material just said a precondition was incorrect. (This question couldn't be answered because it depends on the domain.)

  - *Session 2: training*

    The subject authored the tutorial's procedure with separator drain manifold values rather than cutout valves.

  - *Session 2: 1st procedure*

    The subject was confused about the procedure's description. The test monitor pointed to a description of the procedure's goals.

    The subject was surprised when a menu for the operator name did not appear the second time the subject performed operator's action (i.e. turning the handle).

    The test monitor had to show the subject how to get to the control door.

    When the subject indicated that he was finished, he was told to test the procedure.

- *Changes*

  The domain attribute "sdm_handle_open" is no longer available to subjects. This attribute interferes with learning, but is needed by Steve for determining that the handle has finished turning.

  Subjects that only use the editor (EC_3) can now add control preconditions directly to steps. Before these subjects had to add the preconditions to a conditional effect. The groups using demonstrations (EC_1 and EC_2) already had this capability.

  Modified the description of the first procedure by adding a paragraph. The paragraph reminded the subject that Diligent only asks for an operator's name once. The second time that the operator's action is seen, Diligent does not ask for the name. In the first procedure, the operator for turning a handle is used multiple times.

- *Subject 3*

  - *Session 1*

    Subject asked if he could play with the system while reading the tutorial. The subject was told to follow the directions.

    The subject thought the procedural representation worksheet questions were confusing.

  - *Session 2: training*

    The subject had problems zooming in with Vista.

    The subject forgot to start testing the tutorial's procedure. The subject then asked questions about options that are only available during testing.

    During the practice problem, the subject asked questions. When asked about preconditions, the subject was told, "whatever you think is best." The subject asked if he should test his procedure and was told yes.

  - *Session 2: 1st procedure*

    The subject was not told that he could write on the sheet containing the procedure's description.

    The subject didn't see the picture identifying the separator drain manifold valves.

  - *Session 2: 2nd procedure*

    The subject was told that he could write on the sheet containing the procedure's description.

    The subject asked about the amount of time left when there were 12 and 5 minutes left.

  - *Session 2: later comments*

    The subject thought Vista was too slow.

    The subject didn't feel that he knew the system well enough to recover from errors.

    The subject tried to turn lights on/off by selecting them with the mouse. (Of course, this did not work.)

- *Subject 4*

  - *Session 1*

    Showed the subject how to zoom in with Vista. Vista sometimes responded a little slowly.

  - *Session 2: training*

    Stopped after finishing the tutorial instead of reading the directions. The subject was told to continue.

  - *Session 2: 1st procedure*

    The subject had problems with inconsistent procedure goals. (The subject used the EC_3 editor.)

  - *Session 2: later comments*

    The subject said that having to spell attribute values was not a problem when using the editor.

    During the experiment, the subject asked if he could ask questions. He was told "no."

- *Changes*

  The second day tutorial now shows experimental groups EC_1 and EC_2 how to give a second demonstration. This helps with error recovery.

  During the experiment, subjects are told to start the procedures from the state shown in the Vista window.

  Sheets with pre-printed statements were created. They are used during training and for preparing subjects for authoring the experiment's procedures.

  Sensing actions are disabled in Diligent. This should not impact subjects because students shouldn't use sensing actions.

  Subjects are now told to test the practice problem. (So far, the subjects have tested it.)

  Limit the review at the start of the second session to 10 minutes.

- *Subject 5*

  - *Session 1*

    The subject was confused about the use of pseudo-steps that represent the procedure's initial and goal states.

    The subject accidently started defining a subprocedure and was told to abort it.

    Sometimes the procedure's graph looks different than what is shown in the tutorial. This confused the subject.

    The subject was a little confused about why causal links and ordering constraints are rejected independently. The subject was told that an author may want an ordering constraint without a causal link when he doesn't want to show the causal link's condition to students.

  - *Session 2: training*

    At start of session, the subject was told to focus on the synopsis and procedural representation worksheet. However, the subject could look at other parts of the tutorial.

    Told the subject to do "whatever you think is best" during the practice problem.

  - *Session 2: 1st procedure*

    The subject had a serious error when he demonstrated the procedure too quickly and experienced the simultaneous actions problem. This hurt the final procedure. The test monitor told him what caused the problem.

    The subject thought that the second stage valve would turn off the first stage light. (The first stage valve turns off the first stage light.)

    In the middle of a demonstration, the subject suspended the demonstration. However, this prevents learning and is undesirable in the experiment.

  - *Session 2: 2nd procedure*

    STEVE did nothing while testing the procedure. The test monitor told the subject to abort the test. The test monitor appears to have made a mistake because the symptoms indicated that procedure was bad and that STEVE could not find any appropriate actions to perform.
  - *Session 2: later comments*

    The subject did not like the procedure descriptions.

- *Changes*

  Changed the color of the control door power on and motor on lights. Before this, subjects were told what color was on and off.

  Disabled Diligent's suspend demonstration command. Subjects should not use this feature.

  The description of the experiment's first procedure was changed. It was made explicit that each alarm light can be turned off by opening the corresponding separator drain manifold valve. This change was made because subject 5 thought that opening the second stage valve would turn off both the first and second stage alarm lights.

- *Subject 6*
  - *Session 1*

    The subject had to be shown how to reset the view of the device with the simulation (i.e. VIVIDS).

    The subject tried to think about plans in terms finite state machines.

    The subject had to be shown STEVE's control panel.
  - *Session 2: training*

    The subject had difficulty specifying the step after which a new step is inserted.

    The subject was told that experiments interacted with the environment.

    In the practice problem, the subject was confused about step specific preconditions and conditional effect preconditions. (The subject had obvious misconceptions during the practice problem.)
  - *Session 2: 1st procedure*

    The subject demonstrated actions too quickly twice. This problem could not be fixed.
  - *Session 2: later comments*

    The subject's nearsightedness caused real problems in training and in using the system.

    The subject was frustrated because Vista was slow and moving around in Vista was difficult. (Subjects don't need to zoom or pan during the experiment.)

- *Changes*

  Created solution for practice problem. The solution allows subjects to verify that they understand how to author.

The description of the experiment's first procedure was changed. It was made explicit that subjects should focus on turning off alarm lights that are red.

The description of the experiment's second procedure was changed. It now says to shut the "two air intake valves" rather than the "air intake valves."

- *Subject 7*

  - *Session 1*

    The subject refused to follow directions. He read the synopsis at the end of the tutorial first.

    The subject was a planning expert that believed that a causal link implies an ordering constraint. The subject didn't care about the representation worksheet's answers.

    Showed the subject how to associate an effect with a step.

    The subject couldn't finished because Vista crashed. The subject's data was reloaded, but testing with STEVE didn't work. (STEVE couldn't be used because Diligent was not providing STEVE with some low-level knowledge.) The subject finished the testing section by reading the tutorial.

  - *Session 2: training*

    The subject was told that a procedure's graph was not updated after the window was opened.

    Showed the subject how to answer questions with Steve's control panel. This was the portion of the first session that was skipped after Vista crashed.

  - *Session 2: 1st procedure*

    Wanted to know about checking the condensation, but the test monitor couldn't say anything.

- *Subject 8*

  - *Session 2: 1st procedure*

    The subject had problems with his procedure and wanted to start over. The subject was told to create a new procedure.

  - *Session 2: later comments*

    The subject didn't understand that the ordering relationships shown in a procedure's graph are not updated. The subject felt that this was Diligent's biggest problem.

    The subject thought that each demonstration should contain only one step. This makes learning preconditions more difficult.

- *Changes*

  Subjects in EC_3 can now only add one step at a time. Before they could, specify the previous step and add several sequential steps. This change removed a menu from editor that is very similar to the Demonstration menu used by EC_1 and EC_2. However, by skipping a menu, the editor is a little simpler to use.

The practice problem solutions for groups EC_1 and EC_2 now say that only one demonstration is necessary.

The description of the experiment's first procedure was changed. The description now mentions the initial state. It said that the motor is turned off, two alarm lights are red and the initial state can be seen in the Vista window.

- *Subject 9*
  - *Session 1*

    The subject had problems with Vista. The subject was shown how to select objects.

    The subject was also shown how to reset Vista's the view of environment.
  - *Session 2: training*

    This is the only subject to skip a day between the two sessions.

    The subject had some problems manipulating Vista.

    The subject had problems with the practice problem, which had to be restarted twice. Because of these problems, the practice problem took 20 minutes rather 10. One problem is that the subject performed actions too quickly and experienced the simultaneous actions problem. Another problem is that the subject closed a window with an X-window command instead of using the button provided for the task. When using the X-window command, the subject ignored a window that warned her about closing a window in that manner.
  - *Session 2: 1st procedure*

    The subject was confused about the procedure's description. He wasn't sure whether he needed to open the valves. He was told that needed to open the valves.
  - *Session 2: 2nd procedure*

    The subject asked if the power had to be turned off. He was told, "yes."
  - *Later comments*

    This is the only subject that tried authoring with subprocedures, which is a topic that was not covered during training.
- *Changes*

  The creation and use of subprocedures was disabled.

  The directions for the experiment's first procedure were changed. It now explicitly says that the valves need to be opened and the motor turned on. This is meant to prevent subjects from thinking that either the valves can be opened or the motor turned on.
- *Subject 10*
  - *Session 1*

    The subject performed actions too quickly at the start and experienced the simultaneous action problem. Afterwards, the subject seemed to have no problems. The subject appeared to be familiar with moving around in Vista.

  - *Session 2: 1st procedure*

    The subject expressed concern about her inability to turn off lights, but subject did eventually figure this out.

  - *Session 2: later comments*

    The subject said that editing was hard, but testing with STEVE was easy.

    The subject was also trying to put in optional steps so that the steps could be performed in different orders. (Presently, this is unsupported.)

- *Subject 11*

  - *Session 1*

    Initially, the subject had problems zooming out too far with Vista.

  - *Session 2: training*

    For the practice problem, the subject was shown how to access causal link information.

  - *Session 2: 2nd procedure*

    The subject was told that the power light is white rather than gray at the start of the procedure.

  - *Session 2: later comments*

    The subject felt that the environment was unusual, and it is was difficult getting used to it.

    The subject didn't realize that experiments would remove dependencies. For this reason, the subject was moved from group EC_1 to moved to group EC_2.

- *Changes*

  The practice problem solution for group EC_1 now lists how experiments correct the plan.

- *Subject 12*

  - *Session 1*

    The subject zoomed in too fast in Vista. The subject was shown how to reset the view.

    The subject was very meticulous when covering the tutorial.

  - *Session 2: 1st procedure*

    Experimented without recomputing ordering relationships.

  - *Session 2: 2nd procedure*

    After the 1st procedure but before starting the 2nd, the subject was told to recompute the ordering relationships after testing.

  - *Session 2: later comments*

    The subject felt that Vista zoomed in or out too fast.

    The subject also didn't think that testing was necessary.

- *Subject 13*
  - *Session 1*

    The subject demonstrated the steps in the wrong order. The subject was told to edit the procedure so that it resembled the tutorial's procedure. While the out of order problem was being discovered, the subject saw the test monitor use menus to identify the problem.

  Session 2: later comments

  The subject said that he did not have any problems with Vista.

- *Subject 14*
  - *Session 1*

    Explained to the subject that the Soar window's "wait2" and "wait3" meant that nothing else was happening.

- *Subject 15*

  Quit after the first session.

- *Subject 16*
  - *Session 1*

    The subject was familiar with STEVE but not Diligent. (The procedures being authored during the experiment would not work in the versions of the environment that the subject had seen.)

Appendix D

How to Use Diligent

This section contains selected parts of the first day's tutorial. It focuses on how to create a procedure, add steps to it and edit it. These are the areas where the three versions of Diligent used in the empirical evaluation differed.

To limit this section's length, some things have not been shown. Things not shown include deriving goal conditions, deriving ordering relationships, experimenting and testing. The chapter and tutorial summaries are also not shown.

Most of the following sections represent the version that was given to subjects who could both demonstrate and experiment. This material is probably identical to the material given to subjects who could demonstrate but not experiment. Section D.3 describes how steps, preconditions and state changes are added by the subjects who could only use an editor.

As mentioned earlier, this thesis uses the term "step relationships" while the tutorial uses the term "ordering relationships." In order to maintain consistency with screen snapshots, the term "ordering relationships" will be used in this appendix.

Because Diligent used a whole suite of software components, it was not feasible to include everything in this document. If you would like to get a copy of the system, please contact.

Center for Advanced Research in Technology for Education
Information Sciences Institute
University of Southern California
4676 Admirality Way, Suite 1001
Marina del Rey, California 90292

D.1 Starting to Specify a Procedure

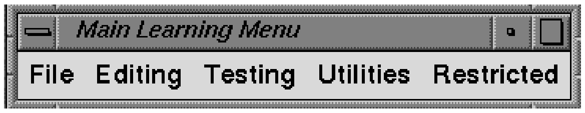


Figure D.1: Main Learning Menu

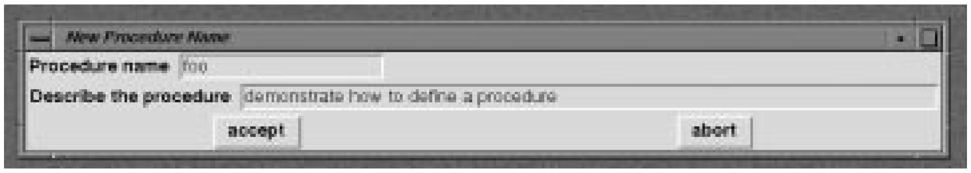
Option	Description
Update existing procedure	Select and change an existing procedure.
Create procedure	Create a new procedure.
Which attributes are used	Allows attributes to ignored when computing ordering relationships.

Figure D.2: Main Learning Menu "Editing" Options

Now that we can manipulate the Vista browser, we are ready to start defining a procedure.

We will be using Diligent's Main Learning Menu. Figure D.1 shows the Diligent's Main Learning Menu and figure D.2 shows the submenu options available under "Editing".

Select the "Create new procedure" option on the Main Learning menu's "Editing" submenu.



Procedure name: foo

Describe the procedure: demonstrate how to define a procedure

Figure D.3: Procedure Description Menu

The menu shown in figure D.3 will appear.

Each procedure has a name, that is used to identify it, and a description, that is given to human students, who are to learn it.

Please enter the procedure name "foo" and the description "demonstrate how to author a procedure".

Indicate that you want to continue defining a procedure by selecting the "Accept" button.

D.2 Demonstrations

This version of the chapter is for when demonstrations are used. The next chapter contains code that was used for evaluation's control group, which was not allowed to demonstrate.

At this point we have started a procedure and given it a name and description.

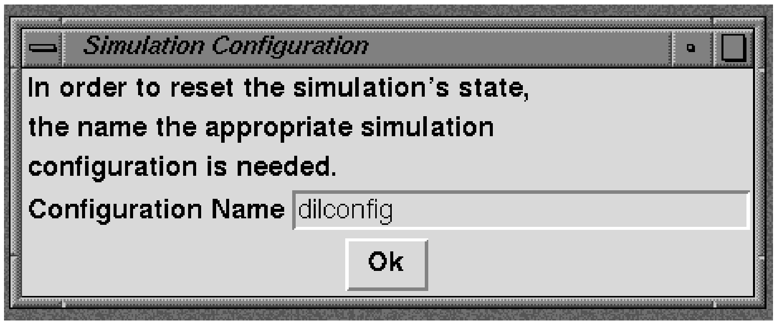
We are now ready to define the procedure's steps. A *step* is another procedure or an action performed in the simulated environment.

We are going specify actions by performing (or demonstrating) them in the Vista window.

D.2.1 Chapter Goals

- Learn how to demonstrate a procedure.
- Learn to provide more than one demonstration.
- Learn about different types of demonstrations.

D.2.2 Setting the Initial Environment State


Simulation Configuration

In order to reset the simulation's state, the name the appropriate simulation configuration is needed.

Configuration Name: dilconfig

Figure D.4: Simulation Configuration Menu

Before we demonstrate the procedure, we need put the environment in the proper initial state.

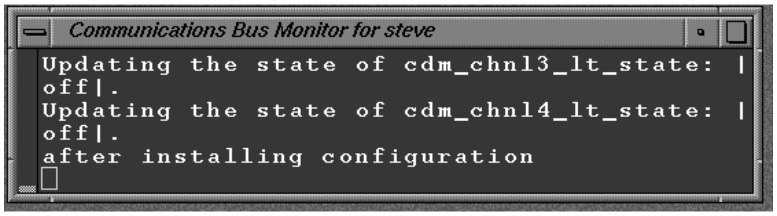
After defining our procedure's name and description, you will see the Simulation Configuration menu (figure D.4), which specifies an initial state for the environment.

Select "Ok" to choose the default configuration.

Resetting the environment takes several seconds. The state has been reset when the text stops scrolling in the Communications Bus Monitor window (figure D.5).

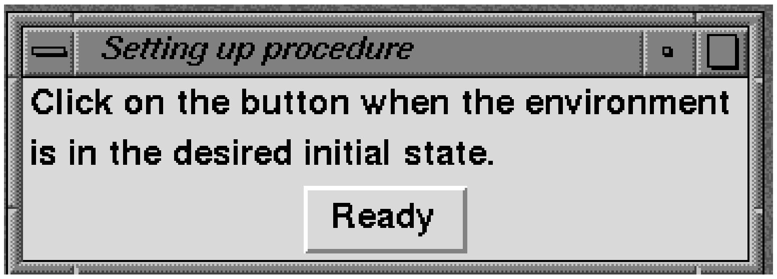
After resetting the environment, you could make additional changes to the environment. Steps will not be added to the procedure until we indicate that we are done making additional changes (figure D.6).

Indicate that we are ready to start adding steps by selecting the "Ready" button figure D.6.



```
Updating the state of cdm_chnl3_lt_state: |
off|.
Updating the state of cdm_chnl4_lt_state: |
off|.
after installing configuration
```

Figure D.5: Communications Bus Monitor Window



Click on the button when the environment is in the desired initial state.

Figure D.6: Additional Environment Changes

D.2.3 Adding Steps

At this point the Demonstration menu will appear (figure D.7). The menu has 3 options that need to be understood.

1. *"Define new subprocedure"* will start the definition of a brand new procedure as a step in the current procedure.
2. *"Insert"* allows use of an existing procedure as a step in the current procedure.
3. *"End demonstration"* will end our demonstration and add the steps we have demonstrated to the procedure.

Before the demonstration, the Vista window should look like figure D.8, and afterwards, it should look like D.9.

Now start the demonstration by toggling the leftmost valve. **Toggle the valve by putting the cursor over it, holding down the SHIFT key, and pressing the left mouse button.**

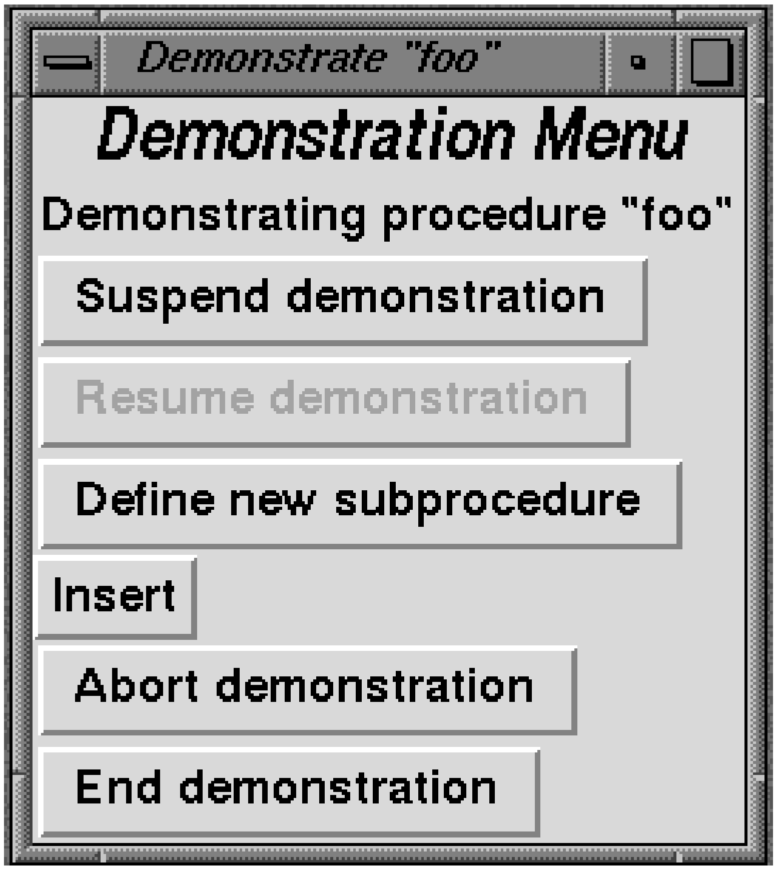


Figure D.7: Demonstration Menu

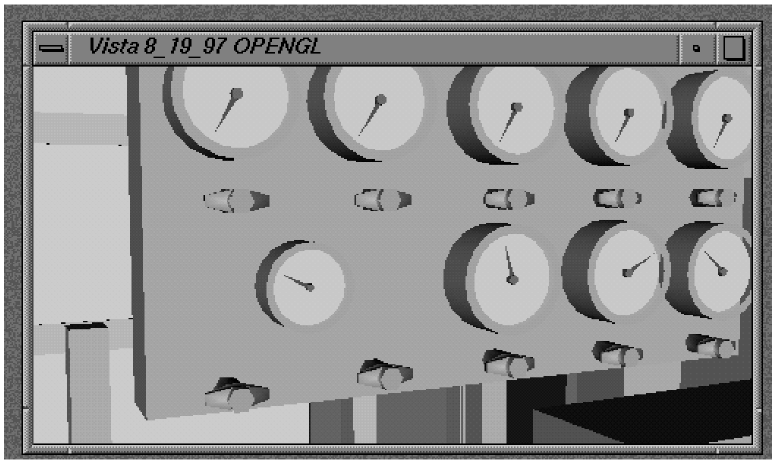


Figure D.8: Environment before Demonstration

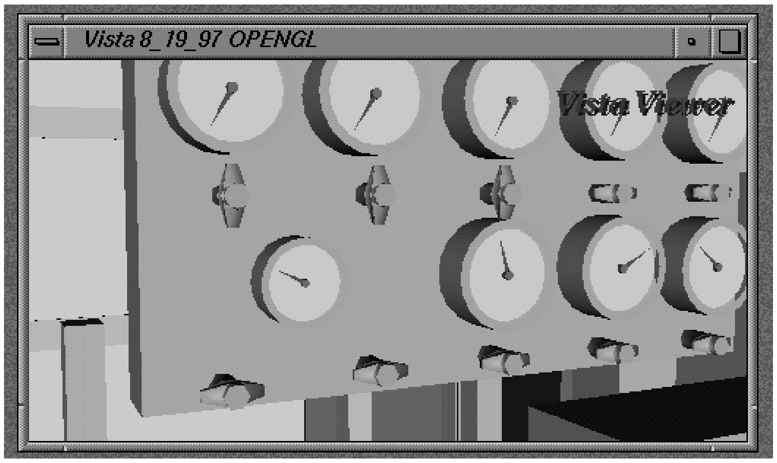
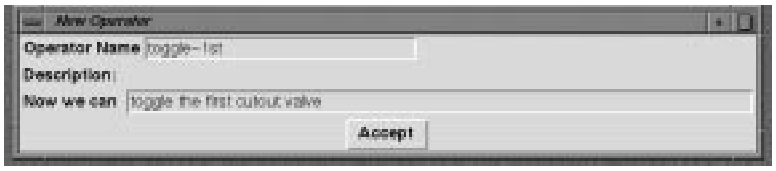


Figure D.9: Environment after Demonstration

D.2.4 Operator Descriptions



Operator Name toggle-1st
Description:
Now we can toggle the first cutout valve

Figure D.10: Operator Description Window

A window will appear that asks for operator information (figure D.10).

What is an operator? Operators describe the preconditions and state changes for actions that are performed in the simulated environment. The preconditions and state changes will be useful for computing the ordering relationships between steps.

The operator's name is used to identify it. **Give the operator the name "toggle-1st".** *The operator's description is given to human students.* **Use the default description, "toggle the first cutout valve." Close the window by selecting "Accept."**

When performing an action, *always make sure Soar has finished processing it.* You can tell that soar is finished when the Soar window looks something like figure D.11. When the processing is finished, "wait2" and "wait3" will be scrolling in the Soar window.

Wait for Soar to finish processing the action.

D.2.5 Add More Steps

To elaborate our example, we will add two more steps to the procedure. This will give you a chance to practice.

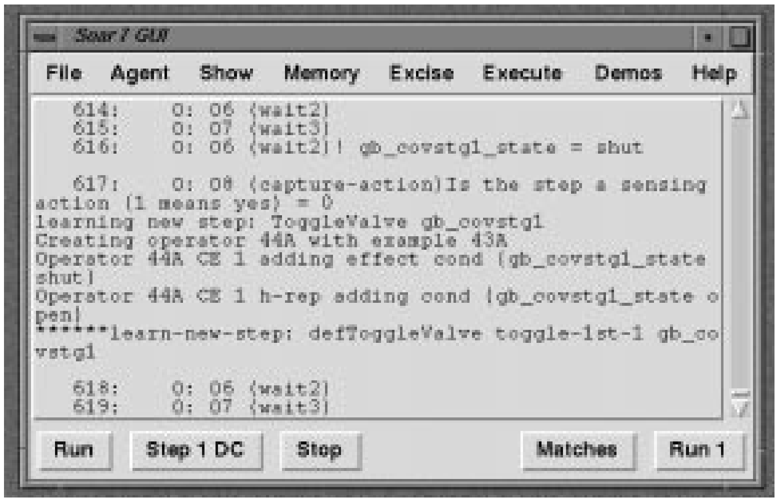
Now manipulate the second valve from the left. **Do this by pressing the left mouse button on the valve while holding down the SHIFT key. Call the operator "toggle-2nd".**

Next, manipulate the third value from the left and call the operator "toggle-3rd". *At this point, the picture in the browser should look like figure D.9.*

D.2.6 End Demonstration

To end our demonstration and add the steps to the procedure, select "End demonstration" on Demonstration menu (figure D.7).

The Demonstration menu will disappear.



```
   614:     O: O6 (wait2)
   615:     O: O7 (wait3)
   616:     O: O6 (wait2)! gb_covstg1_state = shut

   617:     O: O8 (capture-action)Is the step a sensing
action (1 means yes) = 0
learning new step: ToggleValve gb_covstg1
Creating operator 44A with example 43A
Operator 44A CE 1 adding effect cond (gb_covstg1_state
shut)
Operator 44A CE 1 h-rep adding cond (gb_covstg1_state o
pen)
******learn-new-step: defToggleValve toggle-1st-1 gb_co
vstg1

   618:     O: O6 (wait2)
   619:     O: O7 (wait3)
```

Figure D.11: Soar Processing an Action

D.2.7 Additional Demonstrations

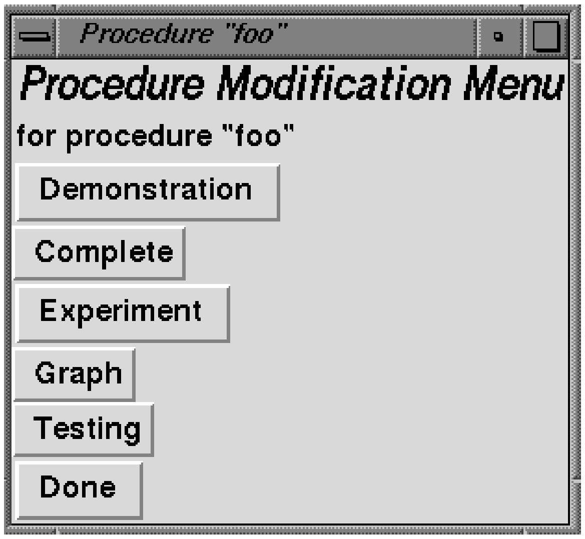
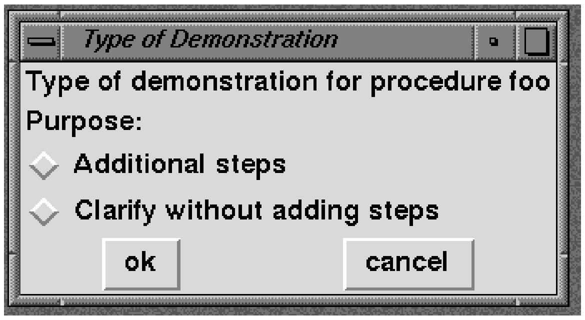


Figure D.12: Demonstration Version of Procedure Modification Menu

After you finish demonstrating a procedure, you can provide additional demonstrations. This is done using the Procedure Modification menu (figure D.12), which is activated when you finish a demonstration.

Start a new demonstration by selecting the “Demonstration” option on the Procedure Modification menu.



Type of demonstration for procedure foo

Purpose:

◇ Additional steps

◇ Clarify without adding steps

Figure D.13: Demonstration Type Menu

A window will appear that asks you to indicate what type of demonstration you want to perform (figure D.13).

- *"Additional steps"* This option allows you to insert additional steps between two steps that are already in a procedure.

- *"Clarify without adding steps"* This option allows you to demonstrate how the environment works without adding any steps. *This type of demonstration helps Diligent discover the preconditions of a procedure's steps.*

  Since Diligent assumes the order that steps are performed is significant, a good heuristic for this type of demonstration is to change the order of the steps as much as possible. For example, our previous demonstration toggled the 1st cutout valve before toggling the 2nd and 3rd cutout valves. A good clarifying demonstration would be to toggle the 3rd cutout valve before toggling the 2nd and 1st cutout valves.

Indicate that you want to give a clarification demonstration by selecting the diamond next to "Clarify without adding steps". Then select "Ok" to continue.

D.2.8 Choosing a Previous Step

Once a procedure has some steps, you need to specify which existing step precedes the first step in a new demonstration.

Figure D.14 shows how the previous step is specified. The upper window contains a graph that shows the order of execution for the procedure's existing steps. The lower window allows you to specify the previous step.

Cancel the demonstration by selecting the "Cancel" button in the lower window. Also close the graph's window by selecting "Ok".

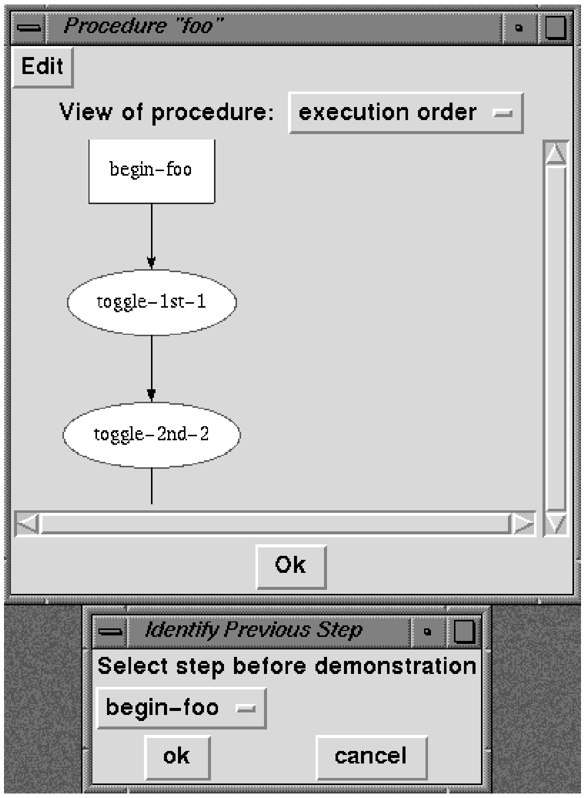


Figure D.14: Previous Step Menu

D.3 Adding Steps to a Procedure

The previous chapter discussed how to demonstrate a procedure. This chapter describes how to add steps to a procedure using only an editor.

At this point we have started a procedure and given it a name and description.

We are now ready to define the procedure's steps. A *step* is another procedure or an action performed in the simulated environment.

D.3.1 Chapter Goals

- Learn how to add steps to a procedure.
- Learn how to associate operator effects with a step.
- Learn how to define operator effect preconditions and state changes.

D.3.2 Adding Steps

After defining our procedure's name and description, you will see the *Procedure Modification menu (figure D.15), which is the main menu for modifying a procedure.*

To add steps to the procedure, select the Procedure Modification menu's "Add a step" option.

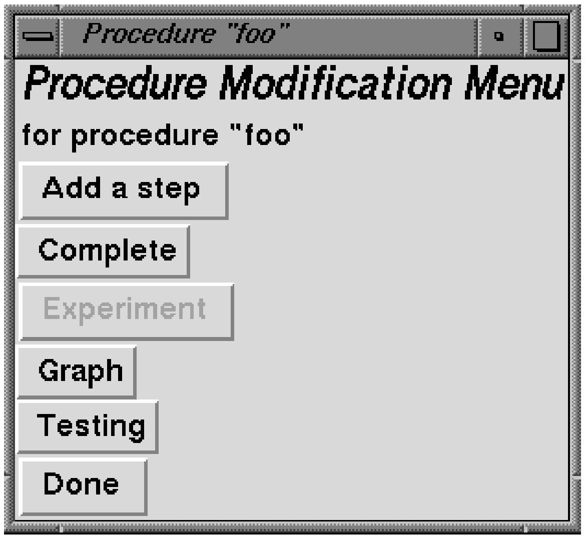


Figure D.15: Manual Editor Version of Procedure Modification Menu

D.3.3 Choosing a Previous Step

Before we can add a step, we need to specify which existing step goes *before* the new step. Figure D.16 shows the windows that help you specify the previous step.

The upper window in figure D.16 contains a graph that shows order of execution for the procedure's existing steps. Initially, there are two steps, which indicate the procedure's beginning and end.

The lower window in figure D.16 allows you to specify the previous step. You could change the previous step by selecting the box containing "begin-foo".

Since the procedure is new, "begin-foo" has to be the previous step. **Agree to continue adding a step by selecting "Ok" in the lower window.**

Also close the graphical view of the procedure by selecting "Ok" in the upper window.

D.3.4 Selecting an Action

The Action Selection menu will appear (figure D.17). The menu describe that the actions that can be added to the procedure.

We want to toggle the first cutout valve. **Select "toggle the first cutout valve." Then approve the action by selecting "Ok".**

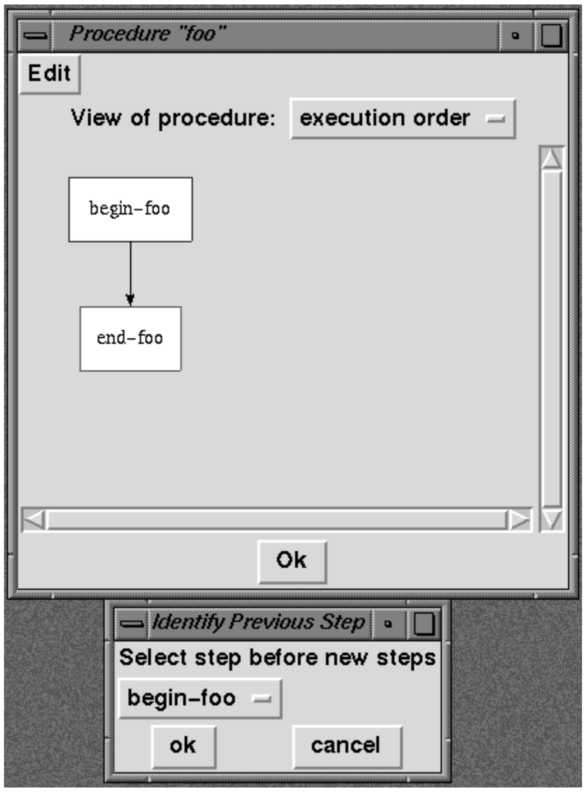
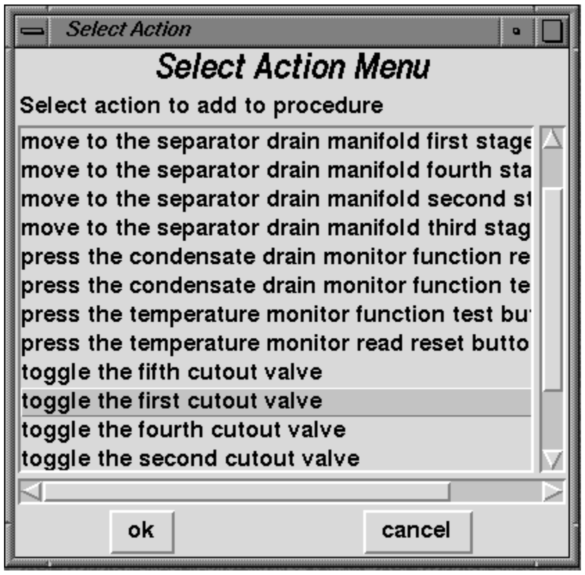


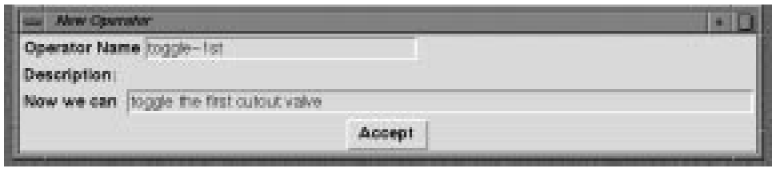
Figure D.16: Previous Step Menu



move to the separator drain manifold first stage
move to the separator drain manifold fourth sta
move to the separator drain manifold second st
move to the separator drain manifold third stag
press the condensate drain monitor function re
press the condensate drain monitor function te
press the temperature monitor function test bu
press the temperature monitor read reset butto
toggle the fifth cutout valve
toggle the first cutout valve
toggle the fourth cutout valve
toggle the second cutout valve

Figure D.17: Action Selection Menu

D.3.5 Operator Descriptions



New Operator

Operator Name: toggle-1st

Description:

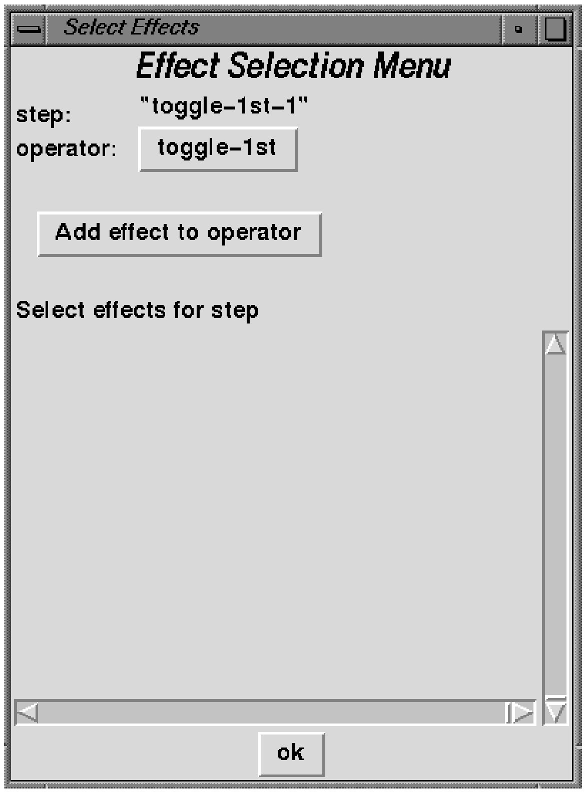
Now we can: toggle the first cutout valve

Figure D.18: Operator Description Window

A window will appear that asks for operator information (figure D.18).

What is an operator? Operators describe the preconditions and state changes for actions that are performed in the simulated environment. The preconditions and state changes will be useful for computing the ordering relationships between steps.

The operator's name is used to identify it. **Give the operator the name "toggle-1st".** *The operator's description is given to human students.* **Use the default description, "toggle the first cutout valve." Close the window by selecting "Accept."**



Select Effects

Effect Selection Menu

step: "toggle-1st-1"

operator: toggle-1st

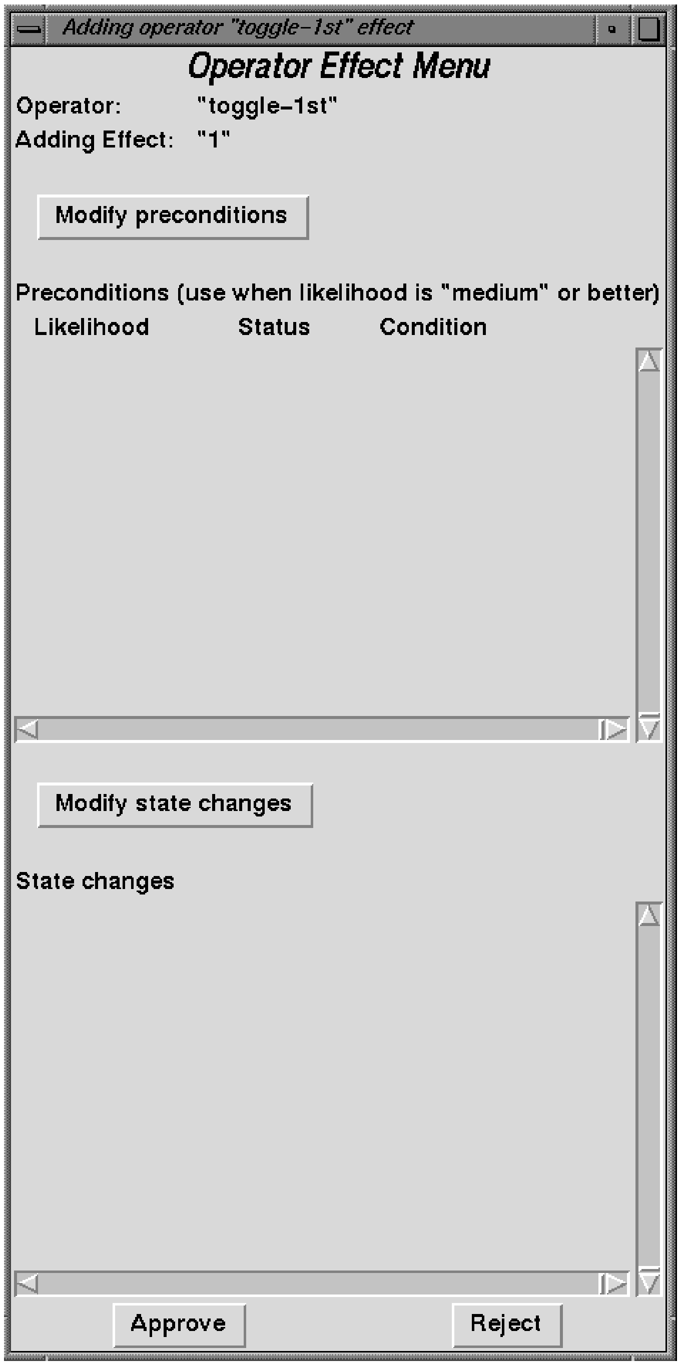
Figure D.19: Effect Selection Menu Before Effects Defined

D.3.6 Selecting Operator Effects

When adding a step not only does the action need to be associated with an operator, but the step must also be associated with some of the operator's effects.

The Effect Selection menu will appear (figure D.19). Unfortunately, the new operator has no defined effects.

Define an effect by selecting "Add effect to operator".



Operator Effect Menu

Operator: "toggle-1st"

Adding Effect: "1"

Preconditions (use when likelihood is "medium" or better)

Likelihood Status Condition

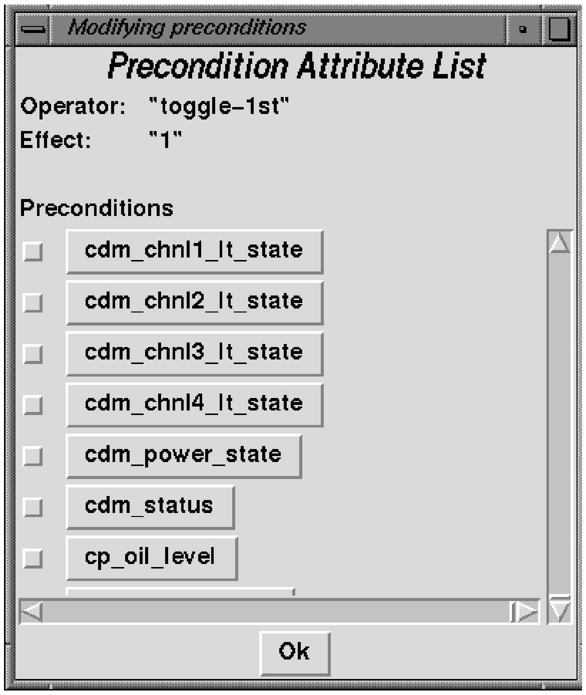
State changes

Figure D.20: Initial Operator Effect Menu

D.3.7 Adding Operator Effects

The Operator Effect menu will appear (figure D.20) for operator “toggle-1st”’s first effect.

Let us first add some preconditions by selecting “Modify preconditions”, which allows us to add, delete and modify the effect’s preconditions.



Precondition Attribute List

Operator: "toggle-1st"
Effect: "1"

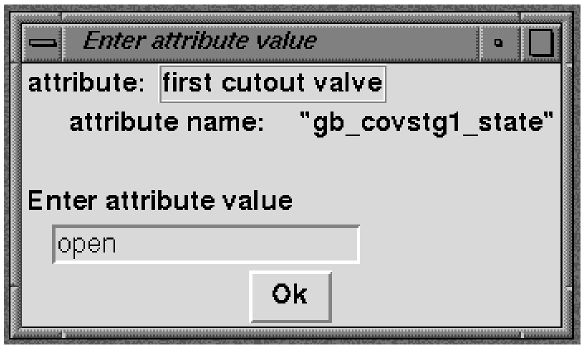
Preconditions

☐ cdm_chnl1_lt_state
☐ cdm_chnl2_lt_state
☐ cdm_chnl3_lt_state
☐ cdm_chnl4_lt_state
☐ cdm_power_state
☐ cdm_status
☐ cp_oil_level

Figure D.21: Precondition Attribute List

A window will appear that contains a list of environment attribute names that can be used in preconditions (figure D.21). If an attribute has a defined value for preconditions, the checkbox (little square box) next to the attribute name will be selected.

Scroll down the list and select the checkbox next to the attribute “gb_covstg1_state”.



attribute: first cutout valve
attribute name: "gb_covstg1_state"

Enter attribute value

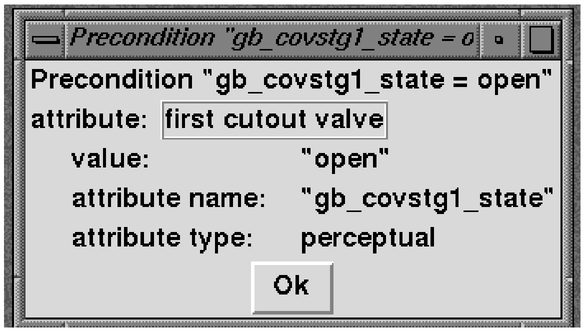
open

Figure D.22: Attribute Value Input Window

The Attribute Value Input window will appear (figure D.22). Figure D.22 shows that attribute “gb_covstg1_state” is described as the “first cutout valve”.

Enter the attribute valve "open" and close the window by selecting "Ok". In the Precondition Attribute list, the square next to attribute's name is now red. Let us look at the precondition that we just defined.

Select the rectangle containing the attribute name ("gb_covstg1_state").



Precondition "gb_covstg1_state = open"
attribute: first cutout valve
value: "open"
attribute name: "gb_covstg1_state"
attribute type: perceptual

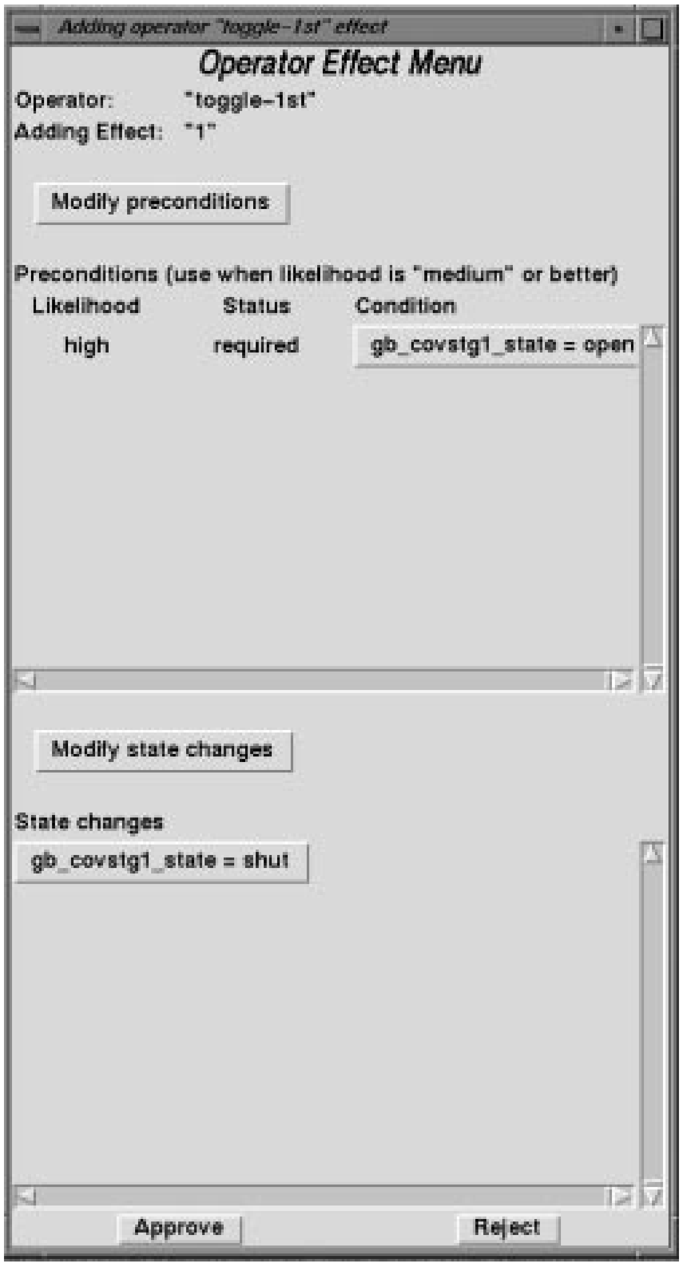
Figure D.23: Precondition Value Window

A window containing information about the precondition will appear (figure D.23). The attribute's description is "first cutout valve," and the attribute's value is "open".

We now want to go back to the Operator Effect menu. **Close the Precondition Value window and the Precondition Attribute List window by selecting "Ok".**

Now that we are back on the Operator Effect menu, we will add a state change. The process is exactly like that used to add preconditions.

Add a state change to the effect by selecting "Modify state changes", which allows us to add, delete and modify the effect's state changes. Indicate that the attribute "gb_covstg1_state" should have the value "shut". When you are done, close the State Change Attribute List and go back to the Operator Effect menu.



Adding operator "toggle-1st" effect

Operator Effect Menu

Operator: "toggle-1st"
Adding Effect: "1"

Preconditions (use when likelihood is "medium" or better)

Likelihood	Status	Condition
high	required	gb_covstg1_state = open

State changes

gb_covstg1_state = shut

Figure D.24: Updated Operator Effect Menu

At this point the Operator Effect menu should look like figure D.24. One precondition and one state change are now defined.

You should know a couple of things about the Operator Effect menu.

1. *Only preconditions with a "Likelihood" of "high" or "medium" are used. By default the preconditions that you add will have a "high" likelihood.*

2. By selecting the rectangle containing a precondition's "Condition" (e.g. "gb_covstg1_state = open"), you can look at information about the precondition. You can also change the precondition's "Status", which control's its "Likelihood".

3. By selecting the rectangle containing a state change (e.g. "gb_covstg1_state = shut"), you can look at information about the state change.

Now add the effect to the operator by selecting "Approve" on the bottom of the Operator Effect menu.

This returns us to the Effect Selection menu.

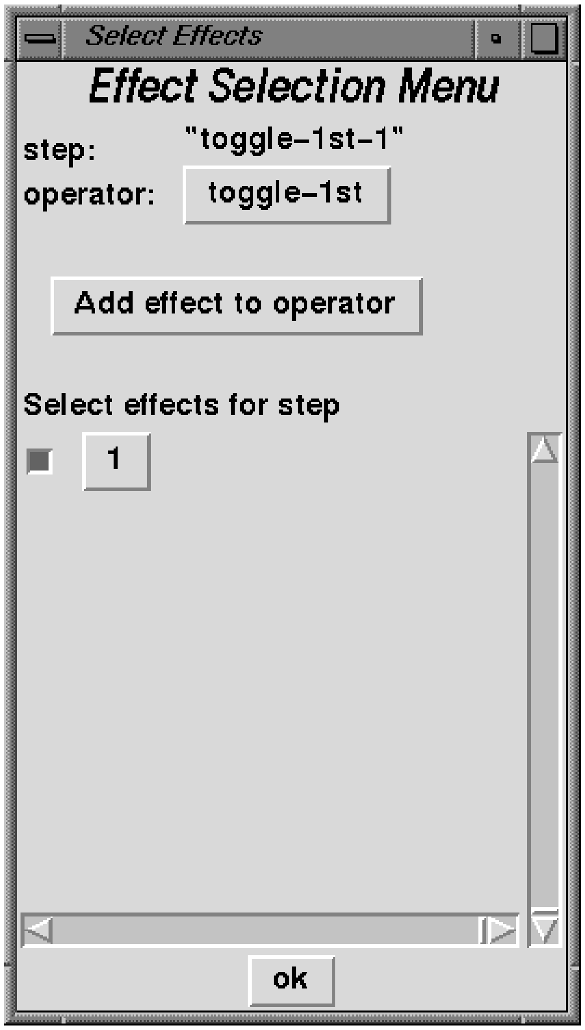


Figure D.25: Updated Effect Selection Menu

D.3.8 Selecting Operator Effect's Revisited

The Effect Selection menu for our first step should now have an effect listed (figure D.25).

Associate the operator's first effect with the step by selecting the checkbox next effect "1".

Now approve the association of step "toggle-1st-1" to operator "toggle-1st"'s first effect by selecting "Ok".

D.3.9 Add a Couple More Steps

To elaborate our example, we will add two more steps to the procedure. This will give you a chance to practice.

After step "toggle-1st-1", add the "toggle the second cutout valve" action, name the operator "toggle-2nd," and have attribute "gb_covstg2_state" change its value from "open" to "shut." "open" is the precondition value, and "shut" is the state change value.

After step "toggle-2nd-2", add the "toggle the third cutout valve" action, name the operator "toggle-3rd," and have attribute "gb_covstg3_state" change its value from "open" to "shut."

D.4 Editing a Procedure

In this chapter will we explore how to edit the objects associated with a procedure.

D.4.1 Chapter Goals

- For objects associated with a procedure,
  - Learn how to examine and modify them.
  - Gain familiarity with their menus.
- Learn about ordering relationships (i.e. causal links and ordering constraints). (See section D.4.11 on page 363).
- Modify our example procedure in preparation for testing.

D.4.2 Review: Reaching the Procedure Modification Menu

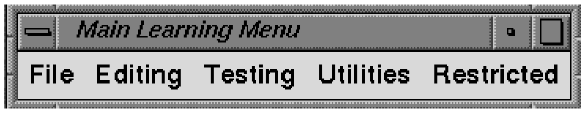


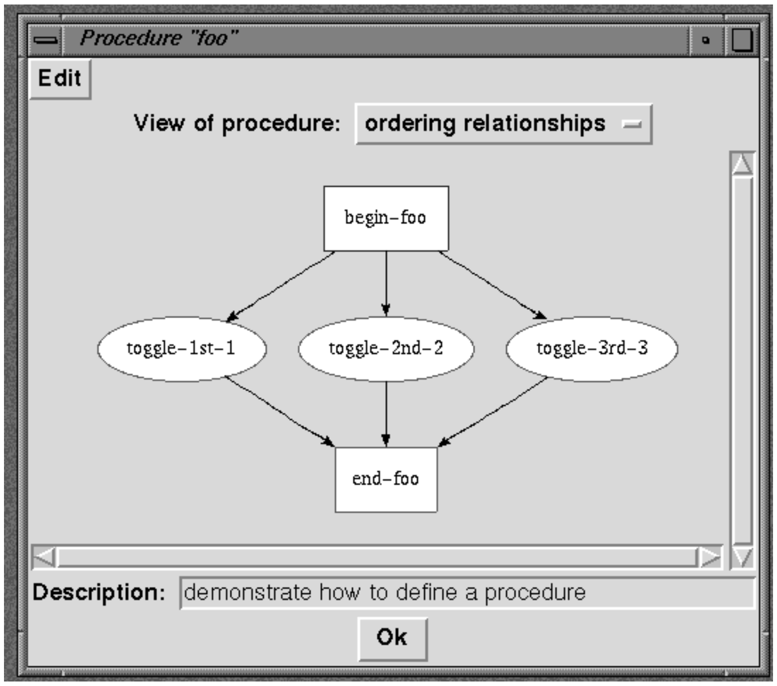
Figure D.26: Main Learning Menu

The Main Learning menu's (figure D.26) "Editing" submenu allows you to access a procedure's Procedure Modification menu.

For an existing procedure, select "Update existing procedure", and a list of procedures appears. Select the name of a procedure and then select "Ok". This will open a Procedure Modification menu for the selected procedure.

Do nothing, the Procedure Modification menu is visible for procedure "foo".

D.4.3 Procedure Graphs



Description: demonstrate how to define a procedure

Figure D.27: Procedure Graph from “Ordering relationships”

A Procedure graph presents the steps in a plan as nodes in a graph and allows you to access data for individual steps.

Create a graph of our procedure by selecting the “Graph” button on the Procedure Modification menu and choosing “Ordering relationships”.

The “ordering relationships” Procedure graph of our procedure is shown in figure D.27. The rectangles “begin-foo” and “end-foo” represent the beginning and end of the procedure. The ovals represent the three steps we specified. The arrows represent ordering relationships between pairs of steps.

The procedure’s initial state is represented as state changes caused by the procedure’s start step (“begin-foo”), and the procedure’s goals are represented as preconditions of the procedure’s end step (“end-foo”) .

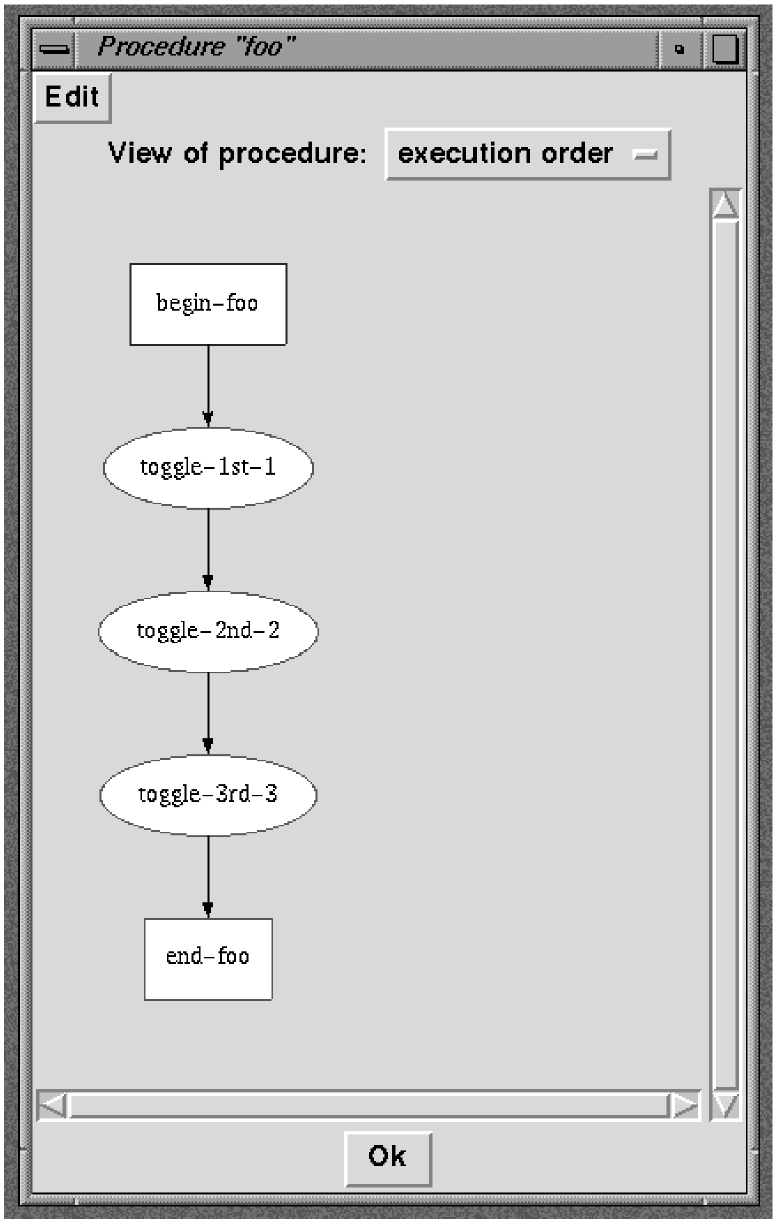
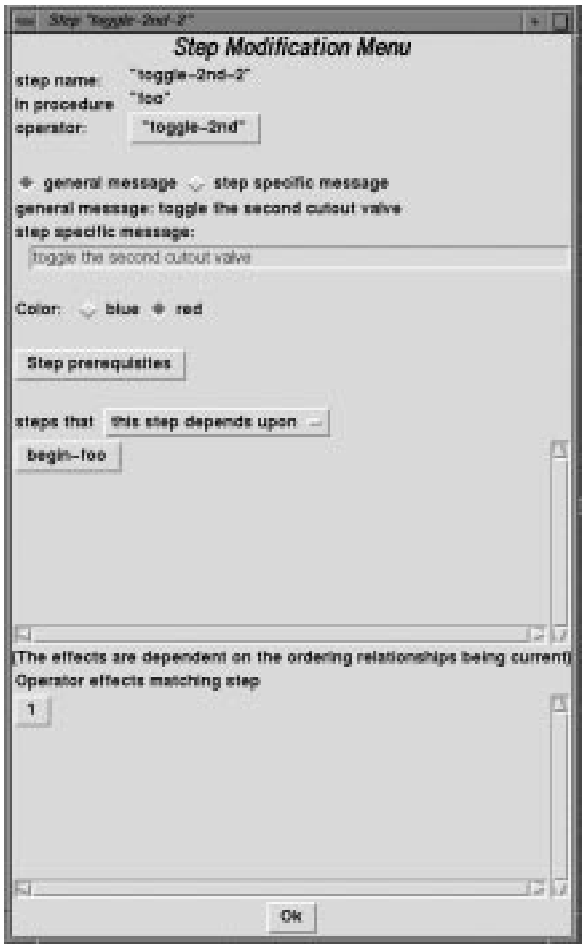


Figure D.28: Procedure Graph showing “execution order”



Step "toggle-2nd-2"

Step Modification Menu

step name: "toggle-2nd-2"
in procedure "foo"
operator: "toggle-2nd"

[✓ general message] step specific message
general message: toggle the second cutout valve
step specific message:
toggle the second cutout valve

Color: blue [✓ red]

steps that: this step depends upon
begin-foo

(The effects are dependent on the ordering relationships being current)
Operator effects matching step
1

Figure D.29: Step Modification Menu

Switch to an execution order view of the procedure by selecting the box containing "ordering relationships" and choosing "execution order".

The "execution order" Procedure graph of our procedure is shown in figure D.28. The arrows order the steps in the sequence that we specified when we added them to the procedure.

D.4.4 Looking at a step

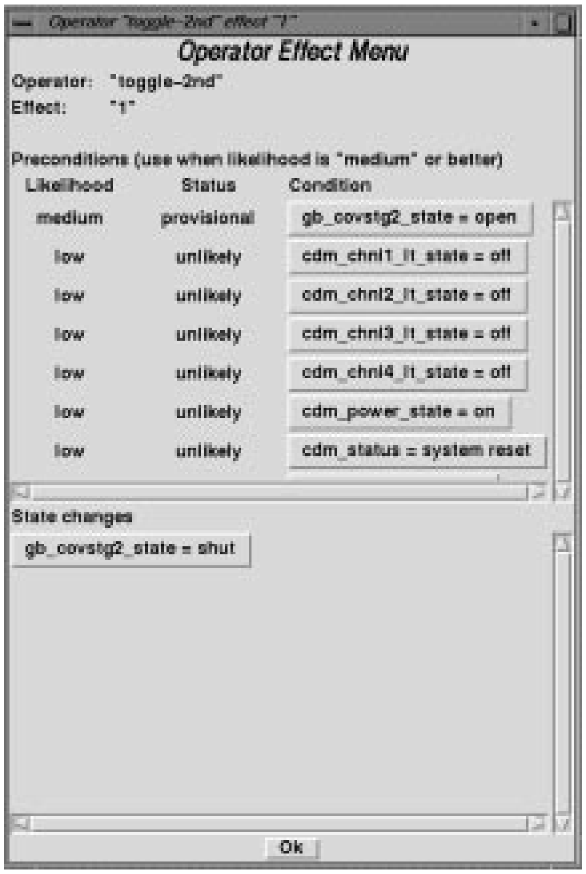
The Step Modification menu allows you to examine and modify objects associated with a step.

Bring up the Step Modification menu for step "toggle-2nd-2" by moving the cursor over the oval containing "toggle-2nd-2". When the oval's outline changes color (becomes black), press the left mouse button.

The Step Modification menu for step "toggle-2nd-2" is shown in figure D.29. The step's operator ("toggle-2nd") associates an action in the environment to the step's effects. This step produces the operator's first effect ("1"). Each effect associates a set of preconditions that must be true before the step to a set of state changes that result from executing the step.

Edit operator “toggle-2nd”’s first effect by selecting the square that says “1”.

D.4.5 Operator Effect Menu


Operator "toggle-2nd" effect "1"

Operator Effect Menu

Operator: "toggle-2nd"
Effect: "1"

Preconditions (use when likelihood is "medium" or better)

Likelihood	Status	Condition
medium	provisional	gb_covstg2_state = open
low	unlikely	cdm_chnl1_lt_state = off
low	unlikely	cdm_chnl2_lt_state = off
low	unlikely	cdm_chnl3_lt_state = off
low	unlikely	cdm_chnl4_lt_state = off
low	unlikely	cdm_power_state = on
low	unlikely	cdm_status = system reset

State changes

gb_covstg2_state = shut

Figure D.30: Operator Effect Menu

The Operator Effect menu maps a set of state changes caused by an action in the environment to a set of preconditions.

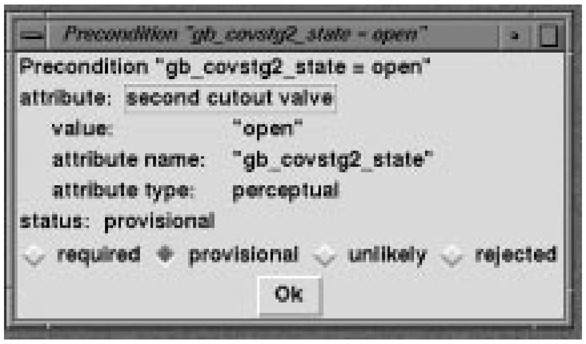
The Operator Effect menu for the first effect of operator "toggle-2nd" is shown in figure D.30.

You should know a couple of things about the menu.

1. The area at the top of the menu describes *preconditions*, which are attribute values that need to be true before the operator's action is performed.
2. *Only preconditions with a "Likelihood" of "high" or "medium" are used.*
3. By selecting the rectangle containing a precondition's "Condition" (e.g. "gb_covstg2_state = open"), you can look at information about the precondition. *You can also change the precondition's "Status", which control's its "Likelihood".*
4. The bottom of the menu lists state changes produced by the effect. *State changes* are the values of attributes after the operator's action is performed.

5. By selecting the rectangle containing a state change (e.g. "gb_covstg2_state = shut"), you can look at information about a state change.

D.4.6 Precondition Window



Precondition "gb_covstg2_state = open"
attribute: second cutout valve
value: "open"
attribute name: "gb_covstg2_state"
attribute type: perceptual
status: provisional
required [✓ provisional] unlikely rejected

Figure D.31: Precondition Window

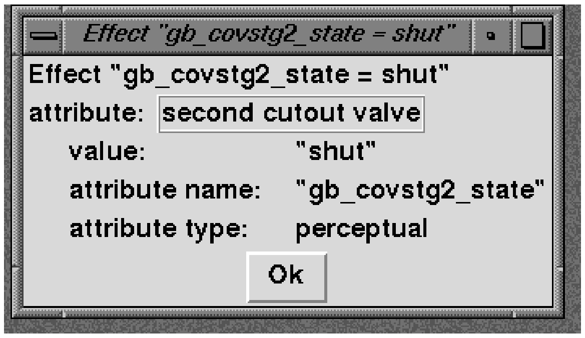
Using the Operator Effect menu, look at a precondition by selecting the rectangle containing "gb_covstg2_state = open".

The Precondition window describes a precondition for an operator effect. Figure D.31 tells us that the state of the "second cutout valve" needs to be "open" and that the precondition is "provisional", which means that it will be used.

Preconditions are used only when their status is "required", "suspect" or "provisional".[1]

Close the Precondition window by selecting "Ok".

D.4.7 State Change Window



Effect "gb_covstg2_state = shut"
attribute: second cutout valve
value: "shut"
attribute name: "gb_covstg2_state"
attribute type: perceptual

Figure D.32: State Change Window

Using the Operator Effect menu, look at a state change by selecting the rectangle containing "gb_covstg2_state = shut".

[1]In the tutorial, this chapter's summary has a table that describes the various status values. In this thesis, the calculation of status values is described in Section A.3.

The State Change window describes a state change caused by an operator's effect. Figure D.32 tells us that the state of the "second cutout valve" will be "shut".

Close the State Change window by selecting "Ok".

D.4.8 Modifying Preconditions

We will now introduce two preconditions for step "toggle-2nd." The preconditions will help us when we test the procedure.

D.4.8.1 Using the Operator Effect menu

The first precondition is erroneous. It will be identified when we test the procedure.

The precondition is the last precondition in the Operator Effect menu's list of preconditions. The precondition has a likelihood of "none" because the experiments determined that it is unnecessary.

Be aware that the scrollbar next to the preconditions does not indicate the number of preconditions in the list.

Select the precondition with the condition "gb_covstg1_state = shut". In the Precondition window, set the status to "required".

Select "Ok" to close the Operator Effect menu.

D.4.8.2 Using the Step Prerequisites menu

The next precondition that we will specify is not required to perform the step. Instead, the precondition is used to control when the step is performed.

Operator effects are inappropriate for this purpose because

- Preconditions are automatically eliminated if they are not required by the environment.
- The same effect could be used with several steps.

You can specifying preconditions for controlling when a step is performed using the "Step Prerequisites" menu.

On Step Modification menu for step "toggle-2nd-2", open the "Step Prerequisites" menu (figure D.33) by selecting the "Step Prerequisites" button.

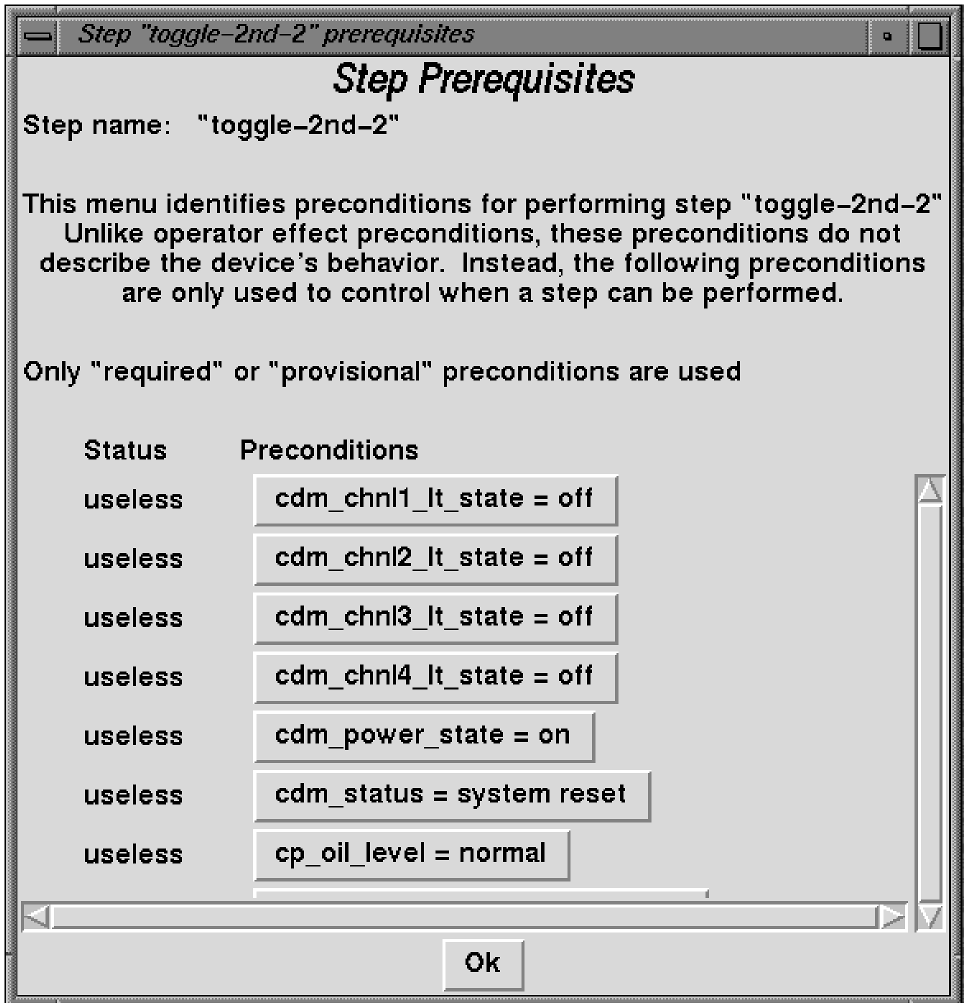
We will specify that the first stage alarm light should be off before performing step "toggle-2nd-2".

Select the precondition for first stage alarm light by selecting the rectangle with the condition "cdm_chnl1_lt_state = off". In the Precondition window, set the status to "required" by selecting the diamond next to "required".

Select "Ok" to close the Step Prerequisites window.

D.4.9 Updated Procedure Graph

After updating the preconditions, we need to close some windows and recalculate the ordering relationships between the procedure's steps.


Step "toggle-2nd-2" prerequisites

Step Prerequisites

Step name: "toggle-2nd-2"

This menu identifies preconditions for performing step "toggle-2nd-2"
Unlike operator effect preconditions, these preconditions do not
describe the device's behavior. Instead, the following preconditions
are only used to control when a step can be performed.

Only "required" or "provisional" preconditions are used

Status	Preconditions
useless	cdm_chnl1_lt_state = off
useless	cdm_chnl2_lt_state = off
useless	cdm_chnl3_lt_state = off
useless	cdm_chnl4_lt_state = off
useless	cdm_power_state = on
useless	cdm_status = system reset
useless	cp_oil_level = normal

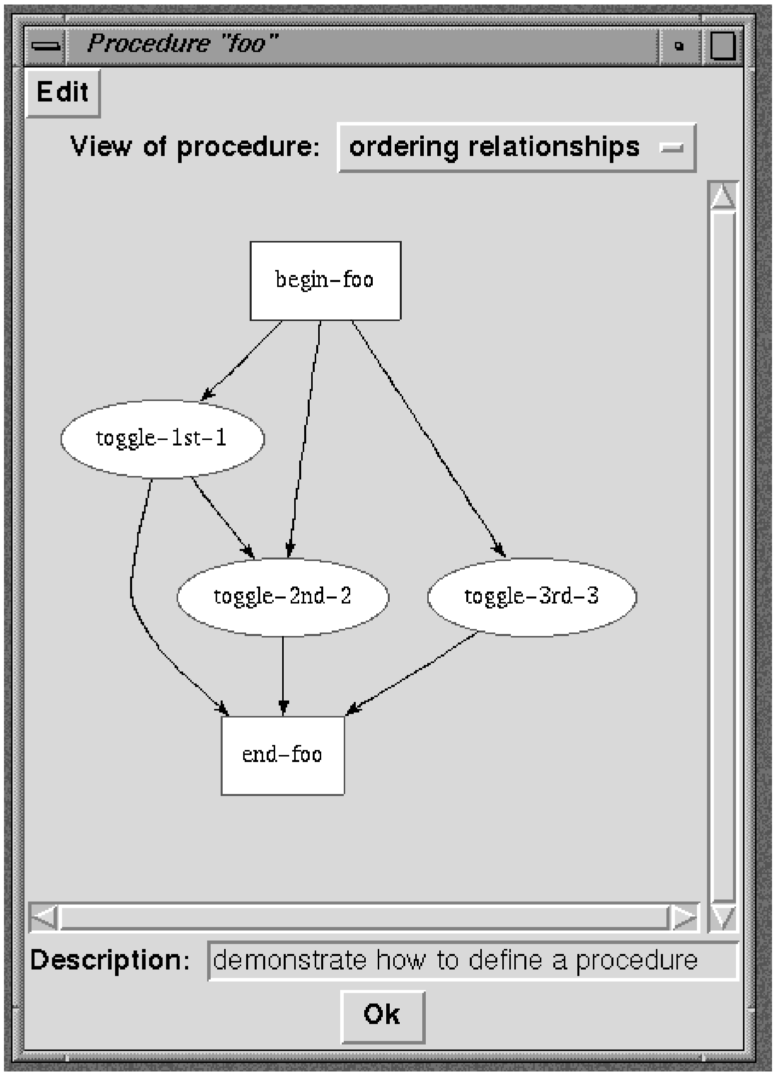
Ok

Figure D.33: Step Prerequisites Menu

Close the Step Modification menu and the Procedure graph by selecting the “Ok” button on the bottom of each menu.

On the Procedure Modification menu, recalculate our procedure's ordering constraints by selecting the “Complete” button and choosing “Derive ordering relationships”.

On the Procedure Modification menu, open up a new Procedure graph, by selecting the “Graph” button and choosing “Ordering relationships”.



View of procedure: ordering relationships

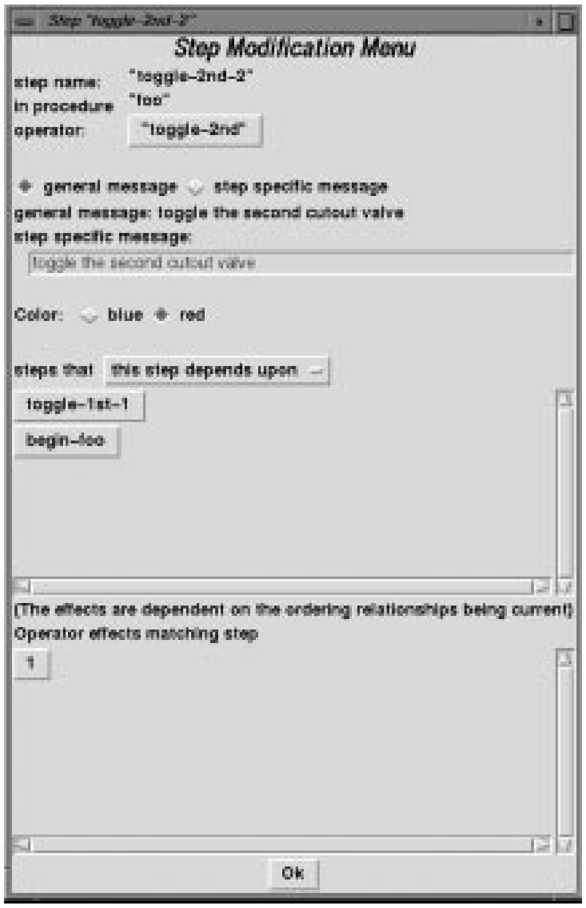
Description: demonstrate how to define a procedure

Figure D.34: Incorrect Procedure Graph

Figure D.34 shows the Procedure graph when operator “toggle-2nd”’s first effect contains the erroneous precondition. You can see the error because the second step (“toggle-2nd-2”) should not depend on the first step (“toggle-1st-1”).

Go to the updated Step Modification menu for step “toggle-2nd-2” by selecting its oval. *Remember to look for a change in color of the oval’s outline.*

D.4.10 Updated Step Modification Menu


Step "toggle-2nd-2"

Step Modification Menu

step name: "toggle-2nd-2"
in procedure "foo"
operator: "toggle-2nd"

[✓ general message] step specific message
general message: toggle the second cutout valve
step specific message:
toggle the second cutout valve

Color: blue [✓ red]

steps that: this step depends upon
toggle-1st-1
begin-foo

(The effects are dependent on the ordering relationships being current)
Operator effects matching step
1

Figure D.35: Step Modification Menu with Error

After the error is introduced, the Step Modification menu looks like figure D.35.

To see dependencies with steps later in the procedure, select “this step depends upon”. *You will see two options “this step depends upon” and “depend upon this step”.* **Choose the “depend upon this step” option.**

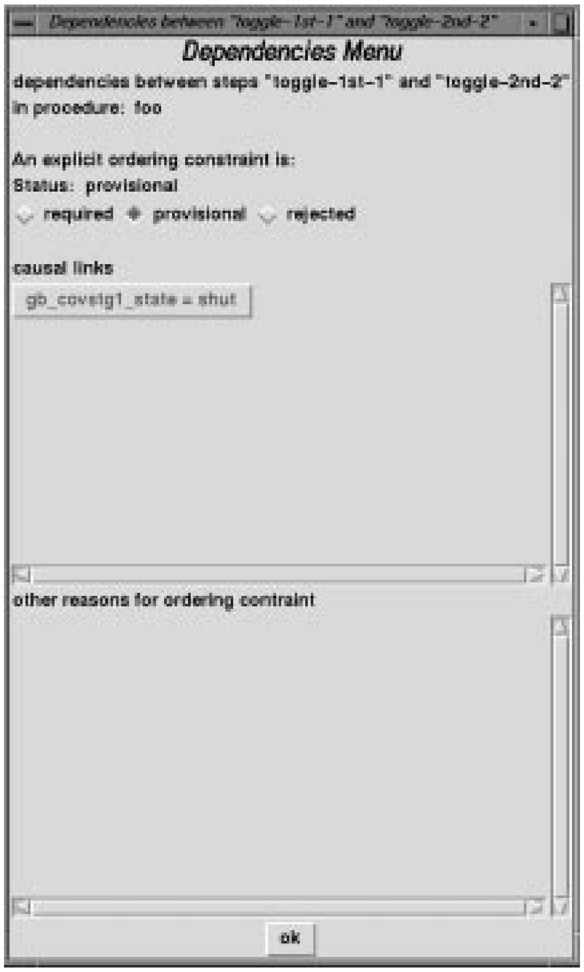
Only “end-foo” will be listed as depending directly on step “toggle-2nd-2”. (The preconditions for step “end-foo” are the procedure’s goals.)

Undo the previous action by selecting “depend upon this step” and choosing the “this step depends upon”.

The menu should now say this step ("toggle-2nd-2") depends on steps "begin-foo" and "toggle-1st-1". ("begin-foo" represents the initial state in which the procedure starts.)

To look at the dependencies between step "toggle-1st-1" and our current step ("toggle-2nd-2"), select the rectangle containing "toggle-1st-1". *This brings up the Dependencies menu for steps "toggle-1st-1" and "toggle-2nd-2".*

D.4.11 Dependencies Menu



Dependencies between "toggle-1st-1" and "toggle-2nd-2"

Dependencies Menu

dependencies between steps "toggle-1st-1" and "toggle-2nd-2"
in procedure: foo

An explicit ordering constraint is:
Status: provisional

required [✓ provisional] rejected

causal links

gb_covstg1_state = shut

other reasons for ordering contraint

ok

Figure D.36: Dependencies Menu

Before we can discuss the Dependencies Menu, we need to define some terms. *Ordering Relationships* are causal links and ordering constraints. A *causal link* is an attribute value caused by one step that is a precondition for a later step. An *ordering constraint* is indicates the relative order for performing a pair of steps.

You want an ordering constraint between the steps when

1. *There is a causal link between the steps.*
2. *The state changes of the latter step interfere with the preconditions of the earlier step.*

Figure D.36 shows dependencies between steps "toggle-1st-1" and "toggle-2nd-2". In figure D.36, notice three things.

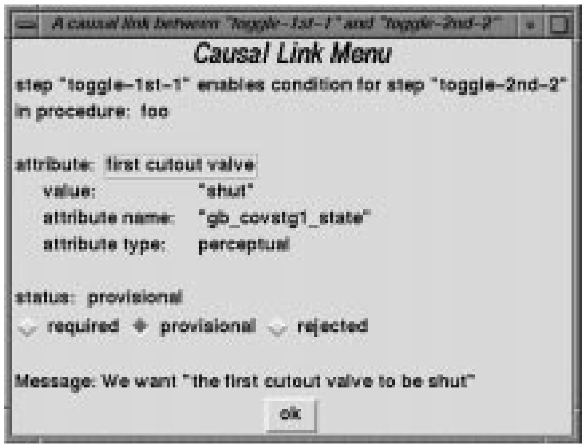
1. Near the top of the menu there is a "provisional" ordering constraint between the two steps. If the diamond next to "rejected" is selected, no ordering constraint will be included in the procedure.

   The ordering constraint says that step "toggle-1st-1" should be performed before "toggle-2nd-2".

2. There is one causal link between the steps with the condition "gb_covstg1_state = shut". This means that step "toggle-1st-1" causes attribute "gb_covstg1_state" to have the value "shut" and that this value is a precondition for step "toggle-2nd-2".

3. The causal link is the only reason for the ordering constraint.

D.4.12 Looking at the Causal Link Menu



A causal link between "toggle-1st-1" and "toggle-2nd-2"

Causal Link Menu

step "toggle-1st-1" enables condition for step "toggle-2nd-2"
in procedure: foo

attribute: first cutout valve
value: "shut"
attribute name: "gb_covstg1_state"
attribute type: perceptual

status: provisional
required [✓ provisional] rejected

Message: We want "the first cutout valve to be shut"

Figure D.37: Causal Link Menu

On the Dependencies menu, look at data for the causal link by selecting the rectangle containing "gb_covstg1_state = shut".

Figure D.37 shows the Causal Link menu. The figure says that there is a causal link between steps "toggle-1st-1" and "toggle-2nd-2" where a state change caused by "toggle-1st-1" is a precondition for "toggle-2nd-2". The state change is that the first cutout valve becomes shut.

The causal link's status is "provisional". Causal links with a status of "rejected" will not be included in the procedure.

Close the open editing windows by selecting their "Ok" buttons. These windows are the Causal Link menu, the Dependencies menu, the Step Modification menu, and the Procedure Graph window.

The Procedure Modification menu should still be open.