Report 83-22
Stanford -- KSL

Scientific DataLink

The Role of Experimentation in Theory Formation.
Thomas G. Dietterich, Bruce G. Buchanan,
May 1983

Heuristic Programming Project
Memo HPP-83-22

May 1983

The Role of Experimentation in Theory Formation

Thomas G. Dietterich and Bruce G. Buchanan
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

Table of Contents

List of Figures

Abstract

This paper presents an analysis of the role of experimentation in theory formation. Experimentation serves three purposes: (a) hypothesis testing, (b) gathering of new data to constrain the theory generator, and (c) manipulation of the external system to reveal its structure. A theory formation system, EG, is described that employs experimentation and observation techniques to develop a theory of the UNIX file system and executive-level file commands. This theory formation task is more complex than previous efforts, and the goal of the project is to determine which existing theory formation methods are applicable and what new methods need to be developed. Previous techniques are reviewed, and none of them are found to be applicable. A new technique, based on controlled experimentation, is described, and a hypothetical trace of EG's execution is presented.

Key terms: Theory formation, generate-and-test, controlled experimentation, exploratory experiments, observation experiments, hypothesis-test experiments, credit assignment, new terms.

1 Introduction

Despite major advances in recent years, Artificial Intelligence models of learning and scientific theory formation are still primitive. This paper describes the EG project—whose goal is to investigate theory formation in a domain that bears a closer resemblance to ordinary scientific domains that previous AI theory formation tasks. We have chosen the task of constructing a theory of the UNIX[1] operating system, because it provides ample opportunities for experimentation under program control. In this paper, we describe this theory formation task and then review previous approaches to theory formation including generate-and-test, version spaces, Meta-DENDRAL, and AM/Eurisko. From this review, we conclude that heuristic search techniques similar to those used in Eurisko are applicable to this domain with modifications to increase the role of experimentation. Three basic functions of experimentation are (a) to test hypotheses, (b) to gather more data for guiding the theory formation process, and (c) to explore the various states of the system being modelled. The paper concludes with a description of the EG program and a hypothetical example of its operation.

---

[1] UNIX is a trademark of Bell Laboratories.

1.1 Description of the theory formation task

The main goal of the EG project is to investigate a large-scale theory formation problem to determine which existing AI techniques can be scaled-up and what new techniques need to be developed. The theory formation task that EG must solve is to develop a theory of the UNIX) operating system by designing, performing, and analyzing experiments with UNIX executive commands. The theory to be developed has four components (see Figure 1): input/output syntax, executive-level commands, system calls, and file-system data structures.

- Input and output syntax. (E.g., Incommand ::= `ln` source dest)
- Abstract programs for each executive command. (E.g., `ln` command, `cp` command)
- System calls. (E.g., `Create-file`, `Delete-file`, `Change-working-directory`)
- File system data structures. (E.g., file directories, protection codes, current working directory)

Figure 1: Four components of EG's theory of UNIX

Input-output syntax includes the syntax of strings typed by the user (commands, file names, arguments, switches, and so on) as well as the syntax of strings printed by the computer (directory listings, prompt characters, error messages, and so on).

The command component of the theory involves describing the behavior of each executive command as an abstract program. There are two basic classes of commands: observation commands, which simply display information about the current state of the system, and manipulation commands, which perform some action on the file system. Manipulation commands typically involve such steps as checking the prerequisites of the action to be performed, printing error messages, requesting confirmation for dangerous actions, and, of course, invoking the appropriate system calls to perform the actions.

The final two components of the theory are closely coupled. The file system is described as a set of mappings and data structures that implement those mappings. The system calls are then defined as operations on these data structures. For example, there is a many-to-one mapping from file names to files, and this mapping is implemented as a table in a directory file. The `Create-file` system call adds a new entry to this table (as well as writing a file on the disk).

This theory is substantially different from any theories developed by previous learning programs in the inter-connectedness of the concepts in the theory. "Theories" formed by Lenat's AM [1976] and Eurisko [1982] programs, for example, are simpler in that they investigate systems of nearly-independent concepts. Each concept can be defined and evaluated without modifying the definition of any other concepts. In EG, however, a change in the definition of a data structure, for example, can have far-reaching effects on the definitions of system calls, commands, and even I/O syntax. .

In defining this particular task, we were guided by an analogy with the way people learn about operating systems. People are rarely taught the details of an operating system. Instead, they read manuals and then conduct experiments. An examination of any operating systems manual will reveal many incompletenesses. For the UNIX manuals we have inspected [Kernighan and McIlroy, 1976; Lomuto and Lomuto, 1983], the syntax and main behavior of each command are well-specified. However, the underlying structure of the file system and the exact behavior of executive commands in unusual situations are very incompletely described. By analogy, EG starts with a fairly complete model of the I/O syntax of UNIX. It is given partial models of the more common executive commands and very little information about the underlying data structures and system calls. It also has a body of general knowledge about computers (e.g., data structures for implementing a many-to-one mapping) and knowledge about the kinds of actions that one would normally want to perform on files and similar data objects.

1.2 Review of previous work in theory formation

Given this knowledge about operating systems in general and UNIX in particular, there is still an immense space of possible theories. We believe that clever experimentation is required in order to search this space efficiently. Before describing the approach that EG employs, let us review existing techniques for searching large hypothesis spaces.

The most naive approach to searching any space is to generate all candidate solutions and test them to see if they are indeed solutions. In this domain, this approach corresponds to generating all theories of UNIX consistent with the starting knowledge base and testing them through experiments. This approach is grossly inefficient, because it does not use the results of hypothesis testing to improve the search as it progresses.

Mitchell's [1978] candidate-elimination algorithm is a technique that completely incorporates information obtained from each test into the generator of possible hypotheses. We believe that the version space approach would not work in this domain, however, because the theory of UNIX is too large to be usefully viewed as a single "concept." The UNIX theory has many nearly-independent subcomponents (e.g., individual commands) that give the space of possible theories a very branchy partial order. Consequently, the candidate-elimination algorithm would not perform well in this space.

The approach employed in Meta-DENDRAL appears initially to be applicable. Meta-DENDRAL is given a set of molecules (with known structure) and their corresponding mass spectra, and its goal is to develop a theory that predicts how molecules will break apart inside the mass spectrometer. It employs what has been termed the "single-representation trick" [Dietterich, et al., 1982] to search the space of possible cleavage rules. First, each data point in each mass spectrum is translated into a set of specific cleavage rules that could have caused that data point to appear. Then syntactic pattern matching methods are applied to generalize these cleavage rules to obtain a small set of general cleavage rules that explain most of the data. An analogous approach in EG's domain would start by gathering training instances of commands and their behaviors. Then the set of maximally-specific UNIX theories that could explain these behaviors would be computed. Finally, syntactic methods would be applied to generalize these theories to obtain a single theory that explained all of the behavior. Unfortunately, this approach is not feasible because the set of maximally-specific theories that explain a given command is too large to compute. Furthermore, it is not obvious which data should be gathered initially to guide the theory formation process. A more dynamic approach appears to be called for.

The AM and Eurisko programs employ a dynamic, opportunistic approach to discovery tasks. It is worthwhile investigating how analogous methods might be applied to EG's task. AM and Eurisko are discovery systems that search a space of possible constructs looking for interesting ones. A direct application of Eurisko to the operating system domain would involve giving Eurisko general knowledge about operating systems along with heuristics for the design of good operating systems and then letting it run to see what interesting systems it discovers. To convert Eurisko into a theory formation system we need to replace the "interestingness" heuristics with techniques that force Eurisko to generate theories that explain the observed behavior of UNIX. This is roughly the approach we are taking. In place of "interestingness" heuristics, EG plans and carries out observation experiments to reveal important structure in UNIX. This structure is then incorporated into the growing theory of the operating system.

1.3 The role of experiments in learning and theory formation

Experimentation (or active instance selection) has been employed in several learning and theory formation systems (see [Buchanan, Mitchell, Smith, and Johnson, 1977]). It provides two basic functions: testing hypotheses and gathering further information to constrain the hypothesis generator.

The simplest learning methods rely only upon the ability of an experiment to test the truth of some proposition and return a YES/NO answer. In pure hypothesis testing, this answer merely indicates that the proposition(or supporting theory) is correct or incorrect. However, it is possible to use this information to help guide the theory generator. For example, as we mentioned above, Mitchell's [1978] version space approach guarantees that the generator will generate only theories that are consistent with the data. Hence,

every YES/NO answer returned from an experiment is completely incorporated into the generator of theories. In this role, experiments can be viewed as providers of new data from which the theory is developed. Even this role has not been developed much in existing learning system. Substantial speedups in learning can be obtained by clever use of experiments to "split the hypothesis space in half" (see [Mitchell, 1978]).

It is possible to exploit YES/NO answers further through credit assignment. If credit and blame can be assigned to individual components of the candidate theory that was tested, then the theory generator can be instructed to modify only the parts of the theory that are still incorrect. There are many techniques for assigning credit and blame. One approach described by Dietterich and Buchanan [1981] is to perform controlled experiments. The generator can generate a set of hypotheses by varying only one component of the hypothesis at a time. Each hypothesis is then tested, and a YES/NO answer is obtained. Because only one component has been varied since the last test, all credit and blame can be assigned to that component for any changes in the overall evaluation of the hypotheses. This strategy only makes sense if it is possible to factor each complete hypothesis into independent subcomponents. If this can be accomplished, then each subcomponent can be tested independently prior to combining the subcomponents into one complete hypothesis. If such a factoring does not exist, then this strategy reduces to exhaustive generate-and-test.

In some domains—including the proposed operating system domain--more than a simple YES/NO answer is available from experiments. In Arthur Samuel's [1967] book-move version of his checkers learning system, for example, each experiment not only tested whether a proposed move was the correct one—it also returned the correct move as a result. This information was then employed to adjust the current hypothesis.

The more information an experiment provides, the more the learning or theory formation system can learn from that experiment. For example, in the LEX learning system, each experiment returns a complete trace of the efforts of the problem solver to solve the given integration problem. A powerful analytic technique has recently been developed for LEX that involves analyzing this trace and attempting to "invert" each integration operator that was part of the optimal solution path. If this inversion can be accomplished, then the evaluation criteria for what it means to solve a problem can be "pushed through" these inverted operators to compute directly the heuristics for each of the operators in the solution sequence. Utgoff [1982] has even employed this technique to compute the definitions of new terms to be included in the generalization language of the system.

We hope to show with EG that similar techniques can be employed in a more complex theory formation task. A very important difference between EG and all previous theory formation systems, however, is that the system being experimented upon is external to the theory formation system. Consequently, it is not possible

to observe directly its internal state. This makes evident a third function of experimentation--it permits the theory formation program to manipulate the external system in order to reveal its internal structure. In the UNIX domain, for instance, experiments can force UNIX into a new state that has never been previously observed. For example, a command to change the current working directory makes a variety of new behaviors observable including changes in the relative names of all files in the system. Once a new state is entered, observation commands can be issued to learn more about that state. From this perspective, it can be seen that EG's task is to explore two spaces: the space of possible operating system theories and the space of possible states of the external system.

1.4 Summary

We are now in a position to state the two basic hypotheses of the EG project:

1. A heuristic search technique similar to that employed in Eurisko is sufficient to discover theories in this domain, and

2. Planned experimentation is essential in large domains in order to avoid combinatorial explosion in the search for theories.

Both of these hypotheses rest on a more fundamental claim: Theory formation can only be accomplished if the system under study can be factored into weakly-interacting components. The essence of heuristic search is to subdivide the overall theory formation task into small subtasks to which heuristics can be applied. The essence of experimentation is to isolate some subcomponent of the system under study in order to observe its behavior (and thus provide guidance to the heuristic search for theories).

This project can be viewed as an attempt to test these hypotheses by developing a new experimentation-based theory formation method.

2 Forming theories about operating system commands

As described in section 1, EG starts with a partial theory of the UNIX file system and incrementally improves this theory by conducting online experiments with a UNIX system over an Ethernet connection. The starting theory of UNIX includes the basic commands for listing file directories, copying files, and deleting files. Also included is an initial theory of the file system and the system calls that manipulate it. This initial theory is incomplete and oversimplified, but it includes enough information to enable EG to manipulate and observe the state of the UNIX system. EG's task is to fill in the missing parts of the theory and deepen the oversimplified parts in order to obtain a complete and accurate theory (at the given level of description).

This section starts by describing the architecture of the EG program. Then a hypothetical example of EG's behavior is presented.

2.1 Program architecture

EG is designed as a set of knowledge sources that interact through a global database and a global agenda. The global database (also referred to as the blackboard) contains a representation of the current theory and any alternatives under consideration. The global agenda is an ordered list of outstanding tasks to perform. An executive routine repeatedly analyzes the agenda, chooses a task, and invokes a knowledge source to perform the task. As the knowledge source runs, it may post new tasks on the agenda.

Tasks fall into two major categories. One kind of task is placed on the agenda when some new information has been posted on the blackboard. This kind of task asks that some knowledge sources be invoked to compute the consequences of this new information and propagate them to other components of the theory. The second kind of task identifies some hole in the theory. It asks that some knowledge sources be invoked to design and execute an experiment to help fill this hole.

Tasks are ranked on the agenda according to their importance. Importance is determined by a collection of heuristic rules such as "If some other task is depending on the results of this task, then this task is important" or "If this task might indicate a serious error in our theory, then this task is important."

From the most abstract viewpoint, the seven knowledge sources in EG can be grouped into two categories: those that generate theories and those that test theories. There are four generator sources, one for each component of the theory:

- TIO—Input/Output theorist
- TC—Command program theorist
- TSC—System call theorist
- TDS—Data structure theorist

Each of these has the task of generating possible modifications to the corresponding component of the UNIX theory. These modifications may involve expanding the theory to include a finer level of detail or revising the theory to correct errors.

There are three knowledge sources that test theories and perform experiments. These are

- The experiment proposer—which examines a problem and attempts to find an experimental method that will help solve it.

- The experiment planner—which develops a specific experiment to perform, and

- The experiment monitor—which performs the experiments and gathers the raw responses from UNIX.

The general behavior of the system is to repeatedly identify a hole in the theory, design and execute an experiment to gather some data, and then invoke appropriate theory-generator knowledge sources to analyze the data and modify the theory.

2.2 An extended example: the ln command

This section presents an extended example of EG as it attempts to understand the `ln` command. This command permits the UNIX user to create an alias name for an existing file. After the command `ln source dest` is executed, both names `source` and `dest` point to the same file, so that changing one file changes the other.

We will start by describing the state of EG's UNIX theory at the point where it begins to work on the `ln` command. Then, we will present a hypothetical sequence of knowledge source invocations.

Suppose that EG has been working for a while on the structure of the file system. It has discovered that the file system is structured as a tree of directories and now it is investigating commands (whose names and syntax it already knows) to see what effects they may have on the file system. One of these commands is the `ln` command. Here is the state of knowledge at this point in EG's researches.

Recall that the theory of UNIX is built of four components: file system data structures, system calls, command programs, and I/O syntax. EG's current theory of data structures states that the UNIX file system is structured as a tree, where each interior node in the tree is a directory and each leaf node is a file. There is a global state variable that points to a particular directory in this tree called the current working directory. A file is designated by giving a path through this tree from the current working directory to the desired file or directory. A downward link in the path is specified by giving the name of the file or directory. An upward link is specified by giving the special name "`..`". Each directory implements a mapping between file names and actual files. This mapping is many-to-one and converts file names into unique global identifiers of files. The mapping is implemented in the directory as a table in which file names and file identifiers are juxtaposed. A unique file identifier is an address that points to the actual string of characters that makes up the file. Each file can be thought of as having a canonical path name that starts at the root of the file structure.

The second component of EG's current theory—the system call component—includes the following primitive actions:

- **create-file(name,contents)**
  Create a new file with the indicated name and contents. This is defined as creating a new unique file identifier, storing the **contents** at that address, and adding a new row **<name, unique identifier>** to the appropriate directory.

- **delete-file(name)**
  Delete the file with the indicated name. This is defined as removing the directory entry for **name**.

- **rename-file(oldname, newname)**
  Change the name of a file. This just changes the name field in the appropriate directory entry.

- **addlink(source, dest)**
  Create a new name (**dest**) for an existing file (**source**). This is defined as adding a new entry to the appropriate directory associating **dest** with the same unique identifier as **source**.

- **filep(name)**
  Check to see if **name** is the name of a file and not a directory.

- **accessiblep(name)**
  Check to see if **name** is defined as either a file or directory.

For each of these primitive actions, there are prerequisites that can be inferred from the current theory of the file-system data structures. For instance, **Create-file** only works if the name does not already exist. **Addlink** requires that the **source** exist and that the **dest** not exist. **Filep** has **Accessiblep** as a prerequisite.

The third component—the command procedure component—includes procedures for the following commands:

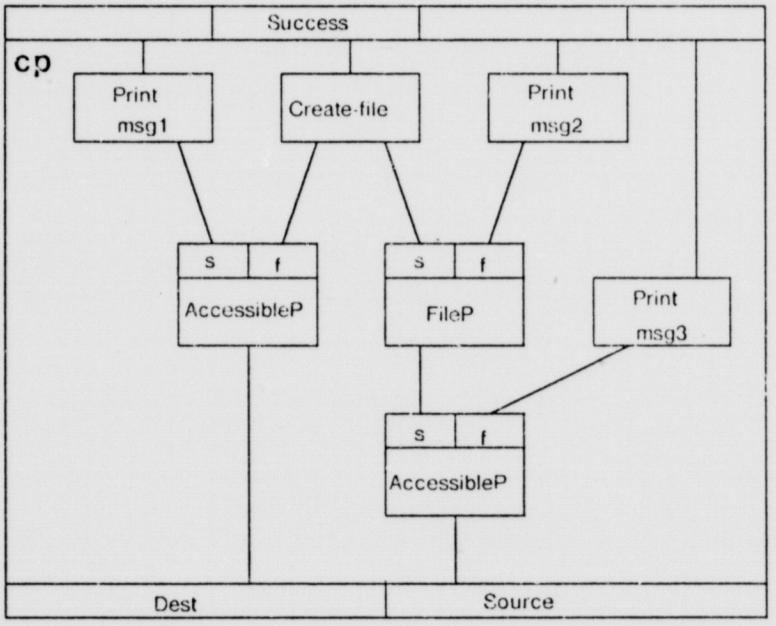
- **ls**
  Print a list of the file names in the current working directory.

- **ls -ld name**
  Print file information about the file or directory indicated by **name**.

- **cp source dest**
  Create a new file **dest** whose contents are copied from **source**.

- **rm source**
  Delete the file **source**.

For each of these commands, EG has an abstract program that defines the command in terms of the system calls given above (see Figure 2). To represent these programs, I am using a slight modification of the programmer's apprentice representation for "deep plans" (see [Rich and Shrobe, 1976]). In this notation, a program is represented as a partially-ordered graph of boxes. Each box denotes a primitive action to be performed. The boxes are linked by control paths that indicate that one box must be performed before another. Conditional boxes have multiple outputs, one for each possible outcome of the action. Although not shown in this simple example, this notation also provides methods for representing iteration and recursion.



msg1: Dest: File exists

msg2: Source is a directory

msg3: Source: No such file or directory

Figure 2: Level 2 theory of the **cp** command.

Finally, the syntax component of EG's UNIX theory includes a complete grammar for file and directory syntax and for command syntax. The following BNF rules show a portion of this grammar.

```
command ::= lncommand | cpcommand
          | othercommand

lncommand ::= ln sp filename sp filename

cpcommand ::= cp sp filename sp filename

filename ::= legalchar
        | filename legalchar
                {repeat up to 13 times}

sp ::= " "
        | sp " "
```

Notice that the syntax for the **ln** command is provided. The syntax for several commands is given to EG at the start. However, the BNF rule for **command** also includes the nonterminal **othercommands**, which has no rules defining it. Hence, EG knows that its list of commands may not be complete. .

Given this body of knowledge, let us suppose that EG has performed an **ln** command and inductively inferred that the purpose of the **ln** command is to perform an **Addlink** operation on its two arguments. Here is one sequence of knowledge-source invocations that might occur.

Step 1. The TC knowledge source is invoked to build a theory of the **ln** command. In other words, it is called upon to develop all possible abstract programs that might implement the **ln** command using the available system calls. TC produces only one theory for **ln**, which is shown in Figure 3. (In my first implementation of this system, the TC builds this theory by backward chaining from the goal of the command.) This theory says that the main step of the command is to perform an **Addlink** operation on two arguments, the source and destination file names. In order to do this, all of the prerequisites of **Addlink** must be checked. Hence, test boxes have been included that check (a) that the source file is accessible, (b) that the source file is a file, and (c) that the destination file is not accessible. Unfortunately, this is not a complete program for the **ln** command. There are three dangling links: cases where TC cannot guess what the **ln** command will do. These correspond to the three cases in which an error is present in the command's arguments. Hence, TC posts three tasks on the agenda—one for each "hole" in the theory.

Step 2. The experiment proposer is invoked to design experiments that resolve the first of these ambiguities—determining what happens when the source file is not a file. The basic plan for such an experiment is to issue an **ln** command for which **filep(source)** is false and observe what happens. Such an experiment will only be valuable if the observation can be made and the output successfully interpreted. Hence, the experiment proposer must find a set of constraints on the arguments to the **ln** command that force control out the desired branch, while at the same time preventing any other, possibly confusing, events from happening.

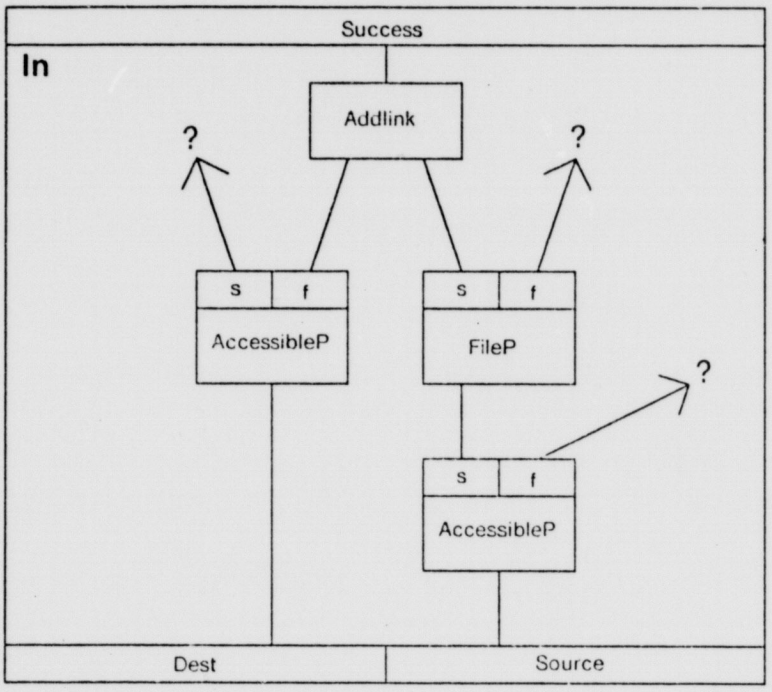


Figure 3: Initial theory of the `ln` command.

I have developed a simple marker-propagation algorithm that can perform this computation. It starts by applying labels to the desirable and undesirable paths in the program. Then, these labels are propagated through the various conditional boxes to determine how each conditional box should be "set." This problem is closely related to the problem of line sensitization in digital circuit test generation [Roth, 1966].

In this particular case, the solution is to choose a source name that is accessible, but that is not a file name (i.e., it is a directory name), and to choose a destination name that is not accessible. An `ln` command with those arguments will not perform the **`Addlink`** operation nor will it send control out any of the other dangling paths. Once the experiment proposer has developed this plan, it adds a task to the agenda suggesting that the experiment planner be invoked to complete the design of this experiment.

Step 3. The experiment planner is invoked to choose specific names for this experiment. If EG does not have an up-to-date model of the current state of UNIX, then some observation commands must first be issued to find out what files exist. Then, the choice of file names is simple. In general, the problem of choosing specific arguments for an experiment is somewhat tricky, since the choices may not be independent of one

another. In such cases, techniques similar to those employed by Stefik [1980] will need to be applied to make some choices and then use those to constrain the other choices. In addition to planning the main step, the experiment planner must also plan the data-gathering commands that are to be issued. In this case, we have some heuristics that tell us that the likely effects of this command are to modify the objects named by its arguments and to display some output. Hence, the following plan is developed:

```
ls -ld d
        Should display
        "drwxr-xr-x 2 xl       32 Apr 20 21:04 d"
ls -ld b
        Should give the error message "b not found"
ln d b
        May print an error message
ls -ld d
        Should display
        "drwxr-xr-x 2 xl       32 Apr 20 21:04 d"
ls -ld b
        Should give the error message "b not found" or else
        directory information about b
```

where **d** is a directory that already exists, and **b** is a non-existing name. Observation actions (the **ls** commands) have been placed before and after the key step (the **ln** command). Notice that the experiment planner computes expectations so that the experiment monitor can determine whether things are "on-course." If **ln** were judged to be a dangerous command, then commands would have been included to create a scratch directory instead of using an existing directory.

Step 4. Now the experiment monitor is invoked to perform the experiment and collect the results. The execution of this experiment produces the following responses from UNIX:

```
% ls -ld d
drwxr-xr-x 2 xl       32 Apr 20 21:04 d
% ls -ld b
b not found
% ln d b
d is a directory
% ls -ld d
drwxr-xr-x 2 xl       32 Apr 20 21:04 d
% ls -ld b
b not found
%
```

(Here are some notes to aid the reader in understanding this trace. The **%** is the UNIX prompt character. The output of the **ls** command--for example **"drwxr-xr-x 2 xl 32 Apr 20 21:04 d"**—is a single line giving information about the file or directory named by its argument. This information includes the protection bits, the owner, the length in bytes, the modification date, and the name.)

Once the experiment monitor has obtained these responses from UNIX, it posts them on the blackboard and adds a task to the agenda suggesting that some knowledge source should interpret this data.

Step 5. The TIO knowledge source interprets these data to mean that the `ln d b` command had no effect except to print the message "d is a directory". This is interpreted as an error message, but, since EG has no ability to understand English, the only thing it can notice about the message is that it mentions the `d` argument (this is typical for UNIX error messages). It is important to note that the problem of interpreting the data is inherently underconstrained. There are two reasons for this. First, the meaning of a string depends on the context in which it appears. A given string appearing in the output, for example, could be the contents of a file, since files can contain arbitrary strings. Or, the string might contain several subfields that denote various objects in the computer, such as protection codes, user names, times and dates, and so on. Second, EG's I/O grammar is never complete, so it is always conceivable that a given string denotes some new class of objects that has never before been encountered by EG. Given this lack of constraint in the parsing process, it is critical that EG be given some knowledge about what to expect. In this context, since we are violating a prerequisite of a primitive action, we are expecting an error message or else some change in one of the command arguments.

Once TIO has interpreted the output, it places this interpretation on the blackboard and adds a new task to the agenda to propagate the consequences of this interpretation to other components of the theory.

Step 6. Hence, the TC knowledge source is invoked to update the command component of the theory in light of the fact that `ln d b` produced only an error message and had no other effects. The `ln` command theory is updated by attaching a `Print` action of the form `Print(concat(source, " is a directory"))` to the dangling link coming from the `FileP` test.

The process of filling in the dangling links of the `ln` command program continues in this way until the plan shown in Figure 4 is obtained. This theory might serve quite well for a long time, but at some point, EG might attempt to create a link that crosses device boundaries. The starting theory of the file system sketched above is incorrect, because it assumes that a directory maps names to unique file identifiers. It turns out that directories map names to file identifiers that are unique for a given device. The combination of the device and the file identifier uniquely designates the file. As a consequence, it is impossible to create links across devices, since the linking operation involves changing only the within-device file identifier.

Suppose that EG attempted the following cross-device `ln` command:

```
% ln filea /tmp/filea
filea: Cross-device link
%
```

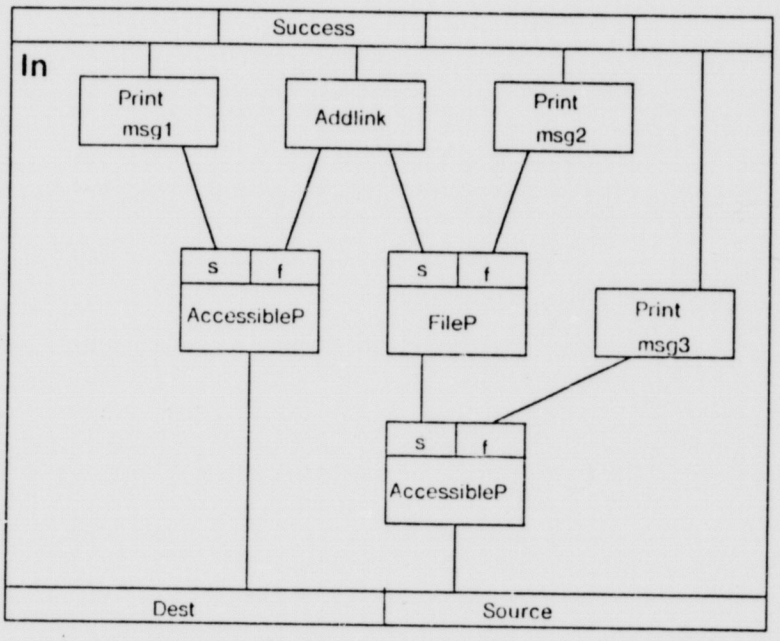


Figure 4: Theory of `ln` after all dangling paths have been observed

The line `file0a: Cross-device link` is an unexpected error message. Here is the sequence of knowledge source invocations that results:

Step 45. The experiment monitor notes immediately that this message violates the expectations for the experiment in progress (i.e., this `ln` command wasn't supposed to produce any output at all). Hence, the execution of the experiment is halted, and TIO is queued to interpret this message.

Step 46. TIO interprets this as an error message (since it is a single line and mentions one of the arguments to the `ln` command).

Step 47. TC updates the theory to include an new conditional box. This box is testing some unknown condition at some unknown point in the `ln` program and printing the error message if the test succeeds. The TC can give this condition box a synthetic name, `Cross-device0`, but it can't be sure what arguments are

being tested. Any data structure in the operating system might conceivably be checked, and EG doesn't even have a complete list of such data structures. Figure 5 shows the possible locations of the `Cross-devicep` check.

TC identifies two problems that must now be solved. First, the exact positions of `Cross-devicep` in the `ln` program need to be determined. Most error messages indicate violated prerequisites, hence, the position of each `Cross-devicep` check will indicate that some primitive action has this condition as a prerequisite. Second, once the positions have been identified, the prerequisite information must be propagated to the higher levels of the theory. These two problems are added to the agenda.

Step 48. The experiment proposer is invoked to work on the first problem. It has two methods for pinning down the position of a test. One is to hold the arguments to `ln` constant while varying the objects that they denote, and the other is to compare the `ln` command with commands that contain some of the same primitive boxes. By applying the first method, the experiment proposer proposes that `ln filea /tmp/filea` be attempted, but with `filea` now being a non-existent file.

Step 49. The experiment planner must figure out a way to accomplish this. For example, EG could simply delete `filea`. However, if the contents of `filea` are valuable, then EG should instead rename `filea`. In this case, EG chooses to delete `filea` using the `rm` command:

```
rm filea
      Should produce no output
ln filea /tmp/filea
      Should either print
      filea: No such file or directory
      or else filea: Cross-device link
ls -ld filea
      Should print filea not found
ls -ld /tmp/filea
      Should print /tmp/filea not found
```

Step 50. Plan execution produces the following output:

```
% rm filea
% ln filea /tmp/filea
filea: No such file or directory
% ls -ld filea
filea not found
% ls -ld /tmp/filea
/tmp/filea not found
%
```

Step 51. TIO interprets this as being the expected output of the `ln` command when the source file does not exist.

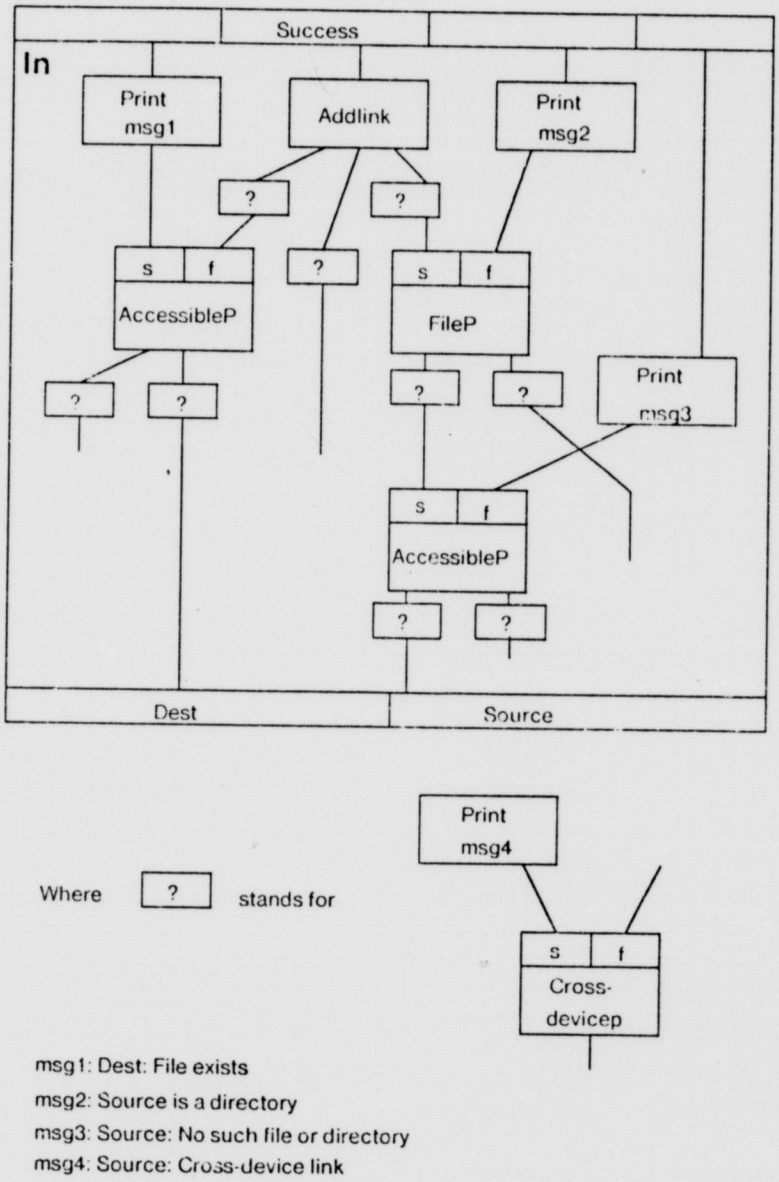
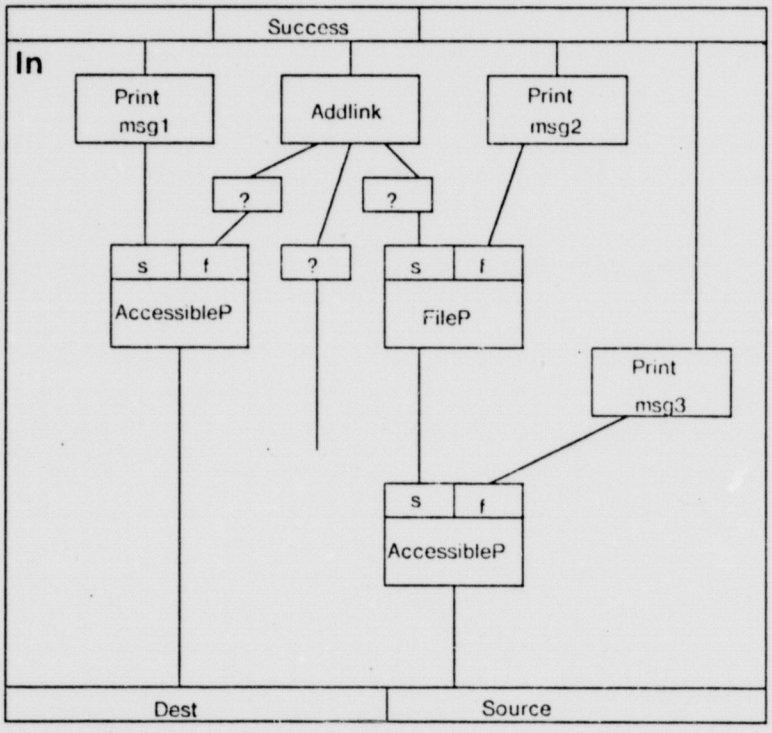


Figure 5: Possible locations of the **`Cross-devicep`** test

Step 52. TC can therefore conclude that, since **`Cross-devicep`** must be checking something besides the existence of **`filea`** and since nothing else changed from one experiment to the next, **`Cross-devicep`** should also have been true in this case. Hence, **`Cross-devicep`** is not a prerequisite of **`AccessibleP`**.

Similar experiments can be conducted to show that **Cross-devicep** is not a prerequisite of the other **Accessiblep** test or the **Filep** test. Hence, **Cross-devicep** must be a prerequisite of the **Addlink** action. The resulting theory is shown in Figure 6. Note that the exact position of the **Cross-devicep** test is still not known.



msg1: Dest: File exists

msg2: Source is a directory

msg3: Source: No such file or directory

Figure 6: Locations of **Cross-devicep** after experimentation

An alternative research strategy would be to compare the **ln** command with the **cp** command. Initially, it appears that these commands have identical prerequisite structures (compare Figures 2 and 3). The only difference is that the final step in the **cp** command is a **Create-file** instead of an **Addlink**.

Step 49 (alternative approach). The experiment proposer suggests that the command **cp filea /tmp/filea** be executed to see if the **Cross-device link** message will be printed. In this experiment (as in the previous step 49), **filea** names an existing file and **/tmp/filea** names a non-existing file.

Step 50 (alternative approach). The experiment planner develops the following experiment:

```
cp filea /tmp/filea
        Should produce no output or else
        the message filea: Cross-device link
ls -ld filea
        Should print
        -rw-r--r-- 1 xl       27 Apr 27 10:44 filea
ls -ld /tmp/filea
        Should print directory line for /tmp/filea
        or else the message /tmp/filea not found
```

Step 51 (alternative approach). The experiment monitor performs this experiment and obtains the following output:

```
% cp filea /tmp/filea
% ls -ld filea
-rw-r--r-- 1 xl       27 Apr 27 10:44 filea
% ls -ld /tmp/filea
-rw-r--r-- 1 xl       27 Apr 27 10:45 /tmp/filea
%
```

Step 52 (alternative approach). TIO interprets the two output lines as directory information.

Step 53 (alternative approach). TC interprets this as meaning that `cp` performed normally, and hence, that `Cross-devicep` was not checked. It follows that `Cross-devicep` is not a prerequisite of `AccessibleP`, `Filep`, or `Create-file`. Hence, it must be a prerequisite of `Addlink`.

Even though the position of the `Cross-devicep` operation has been determined, this does not resolve the question of what it is checking. EG now takes up this problem.

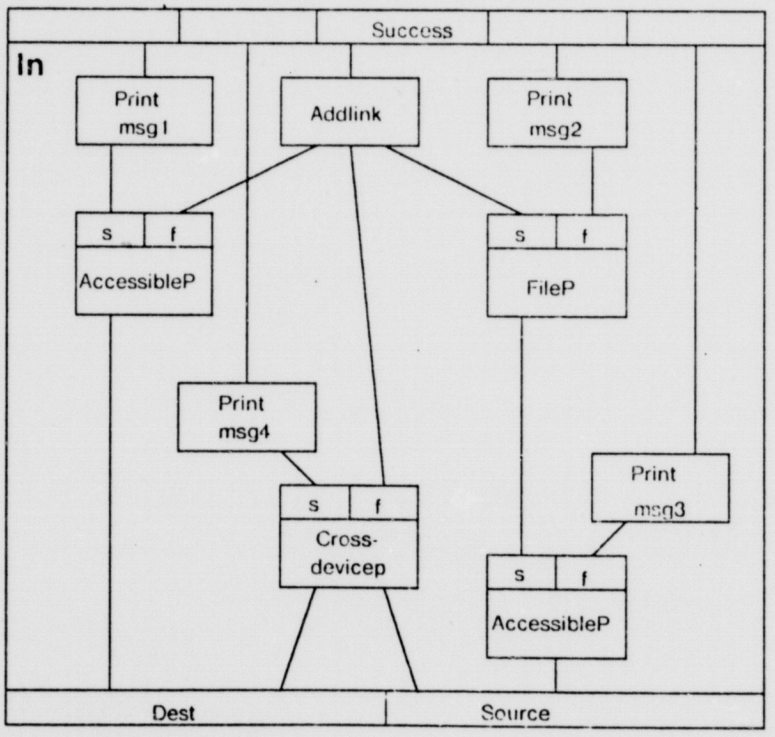
Step 65. The experiment proposer suggests performing several different experiments in an attempt to discover a pattern in the situations in which `Cross-devicep` is true. These experiments vary conditions that might possibly be relevant to `Addlink`, such as the name and location of the source and destination files in the directory structure, the size of the file, the owner of the file, and so on. These experiments are not shown here.

Step 102. Eventually, TSC (the system-call knowledge source) analyzes the data and discovers that `Cross-devicep(s,d)` is true if and only if the first names in the canonical file names of `s` and `d` are different. This turns out to correspond to device boundaries on the UNIX system where these experiments are run.

Step 103. TDS (the data structure knowledge source) attempts to find a change in the data structure theory

that can explain why this would be a prerequisite of **Addlink**. One such change involves altering the definition of the mapping between unique file identifiers and files. In the starting theory, this was a one-to-one mapping represented by juxtaposition. If this mapping is extended to map from the ordered pair **<first-name-of-the-canonical-file-name, unique-identifier>** to files, then **Addlink** can be defined as modifying only the **unique-identifier** field, so that two files must share the same **first-name-of-the-canonical-file-name** in order for the **Addlink** operation to make any sense.

Step 104. Given this revision in the definition of **Addlink**, TC revises the program for the **ln** command to include **Cross-devicep** as a third independent test to check the prerequisites of **Addlink**. Figure 7 shows the final program for the **ln** command.



msg1: Dest: File exists
msg2: Source is a directory
msg3: Source: No such file or directory
msg4: Source: Cross-device link

Figure 7: Final program for the **ln** command

3 Relationship to scientific theory formation

The hypothetical example shown above exhibits some interesting behavior, and one can easily draw parallels with the way in which people learn operating systems. It is worthwhile to ask, however, if EG can tell us anything about scientific theory formation.

Despite the artificial aspects of the operating system domain, we believe that there are several important similarities between this theory formation task and ordinary science.

- The theory being formed is not a single sentence or equation, but rather a description of the operating system at four distinct levels of analysis.
- Interaction with the operating system takes place at the level of strings of characters, which must be interpreted in terms of the current theory. (Observations are "theory-laden" [Hanson, 1958].)
- Data are gathered indirectly through the design, execution, and interpretation of experiments.
- Experiments go beyond simple hypothesis-test experiments—in which a single YES/NO answer is obtained—to include exploratory experiments, in which some new stimulus causes the operating system to enter an unknown state and then observation commands are issued to learn about that state.

Since EG's task resembles a scientific task, we can now ask what EG's methods tell us about scientific methods. In addition to hypothesis testing, EG demonstrates two other important roles for experimentation in science: data gathering and exploration of the system being studied. EG may also shed some light on the origin of theoretical terms. The methods by which EG revises its theory of data structures and system calls can be viewed as a technique for developing new theoretical terms. EG isolates and uses these terms before giving them a precise definition. This technique is analogous to the method employed by Utgoff [1982]. New terms at one level of description are eventually defined as composites of existing terms at a more primitive level.

4 Summary

The goal of the EG research project is to attack a complex theory formation task in order to see what existing methods can be scaled-up and what new methods need to be developed. The task facing EG is to develop a theory of the UNIX operating system that explains the observable behavior of the system at the executive level. Our hypotheses are that (a) no existing methods will work in this domain and (b) manipulation and observation experiments will be critical to controlling the search for theories in this domain.

The EG program is an agenda-driven system in which only a few current hypotheses are maintained. The agenda is a list of problems that require attention including (a) intermediate results whose consequences need to be propagated to other parts of the theory and (b) holes in the theory that need to be filled. Theory-generation knowledge sources perform type-(a) tasks, and experimentation knowledge sources design experiments to solve type-(b) problems. The theory revision process can require that new data structures and system calls be defined in order to make sense of operating system behavior. Implementation of EG has only been under way for a short time.

5 Acknowledgments

This paper owes much to extensive conversations with Jim Bennett on the generate-and-test paradigm for intelligent systems. Specific comments on an early draft of this document have been provided by Jim. The design and development of EG has benefitted from discussions with Mike Genesereth, Saul Amarel, and Lindley Darden. Jock Mackinlay originally suggested the idea of experimenting with operating systems, and Chuck Paulson helped relate the present work to work on digital circuit testing and diagnosis. This research was supported by an IBM graduate fellowship (to TGD), by ARPA grant no. N00039-83-C-0136, by NIH grant no. RR 00785, and by NLM grant number 5-PO1-LM03395.

6 References

Buchanan, B. G., Mitchell, T. M. Smith, R. G., and Johnson, C. R., Jr., "Models of Learning Systems," in J. Belzer, A. G. Holzman, and A. Kent (eds), *Encyclopedia of Computer Science and Technology*, Vol. 11, Marcel Dekker, New York, pp. 24-51, 1977.

Buchanan, B. G., and Mitchell, T. M., "Model-directed learning of production rules," *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (eds.), Academic Press, New York, 1978.

Dietterich, T. G. and Buchanan, B. G., "The role of the critic in learning systems," Rep. No. STAN-CS-81-891, Department of Computer Science, Stanford University. December, 1981.

Dietterich, T. G., London, R., Clarkson, K, and Dromey, G., "Learning and inductive inference," Chapter XIV in Vol. 3 of *The Handbook of Artificial Intelligence*, Cohen, P. R., and Feigenbaum, E. A., (eds.). 1982. Also available as Rep. No. STAN-CS-82-913, Department of Computer Science, Stanford University. May, 1982.

Hanson, N. R., *Patterns of Discovery*, Cambridge University Press, Cambridge, 1958.

Kernighan, B. W., and McIlroy, M. D., "The UNIX Programmer's Manual," Bell Laboratories, 1976.

Langley, P. W., "Descriptive discovery processes: Experiments in Baconian science," Rep. No. CS-80-121, Computer Science Department, Carnegie-Mellon University, 1980.

Lenat, D. B., "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Rep. No. STAN-CS-76-570, Computer Science Department, Stanford University, 1976.

Lenat, D. B., "Eurisko: A program that learns new heuristics and domain concepts," Rep. No. HPP-82-26, Heuristic Programming Project, Department of Computer Science, Stanford University, 1982.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J., *Applications of artificial intelligence for organic chemistry: The DENDRAL project.* McGraw-Hill, New York, 1980.

Lomuto, A. N., and Lomuto, N., *A UNIX* Primer,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

Mitchell, T. M., "Version spaces: an approach to concept learning," Rep. No. STAN-CS-78-711, Department of Computer Science, Stanford University, 1978.

Mitchell, T. M., Utgoff, P. E., Banerji, R., "Learning by experimentation: acquiring and modifying problem solving heuristics," Rep. No. LCSR-TR-31, Computer Science Department, Rutgers University, 1982.

Rich, C., and Shrobe, H. E., "Initial report on a LISP programmer's apprentice," Rep. No. TR-354, AI Laboratory, Massachusetts Institute of Technology, 1976.

Roth, J. P., "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development,* pp. 278-281, October, 1966.

Samuel, A. L., "Some studies in machine learning using the game of checkers. II—Recent progress," *IBM Journal of Research and development,* 11:601-617, 1967.

Stefik, M. J., "Planning with constraints," Rep. No. 80-784, Computer Science Department, Stanford University, 1980.

Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, August, 1982.