# A Toolbox for Analyzing Programs[*]

Scott D. Anderson
David M. Hart
David L. Westbrook
Paul R. Cohen
{anderson, dhart, westy, cohen}@cs.umass.edu

Experimental Knowledge Systems Laboratory
Computer Science Department, LGRC
University of Massachusetts
Amherst, MA 01003-4610

To appear in the International Journal on AI Tools

**Abstract**

The paper describes two separate but synergistic tools for running experiments on large Lisp programs. The first tool, called Clip (Common Lisp Instrumentation Package), allows the researcher to define and run experiments, including experimental conditions (parameter values of the planner or simulator) and data to be collected. The data are written out to data files that can be analyzed by statistics software. The second tool, called Clasp (Common Lisp Analytical Statistics Package), allows the researcher to analyze data from experiments by using graphics, statistical tests, and various kinds of data manipulation. Clasp has a graphical user interface (using CLIM, the Common Lisp Interface Manager) and also allows data to be directly processed by Lisp functions. Finally, the paper describes a number of other data-analysis modules have been added to work with Clip and Clasp.

## 1 Introduction

The systems described in this article, Clip and Clasp were originally developed to aid our empirical, statistical work with an artificial intelligence (AI) program called Phoenix [1] that does online planning and execution in a very complex environment. The statistical work is necessary because the program is so complex that patterns of behavior (and misbehavior) only emerge from many trials in many conditions. As programs become more complex, researchers will increasingly need to turn to simulators, controlled experiments, and statistics to study the behavior of their systems. Clip and Clasp are tools to make such empirical work easier, and although we will describe them with respect to an AI simulator, they are applicable to the statistical study of any large computer program.[1]

We will briefly describe a simulator called TransSim, an AI agent that observes and helps control the simulated environment, and a controlled experiment that the Experimental Knowledge Systems Laboratory (EKSL) ran using TransSim. However, our real purpose in this description is to introduce the two tools that EKSL has developed to aid in running and analyzing experiments of this sort: Clip and Clasp (Common Lisp Instrumentation Package and Common Lisp Analytical Statistics Package).

Clip enables researchers to define experiments in terms of the conditions under which the simulator is to be run and the data to be collected. Clip also helps with the running of the experiment, by looping over all the experimental conditions, running the simulator, and writing the data to files. At that point, a

[*]This research is supported by ARPA/Rome Laboratory under contracts #F30602-91-C-0076 and #F30602-93-C-0100.

[1]As will be described later, Clip integrates with Common Lisp programs and therefore is restricted to programs written in that language. Clasp, on the other hand, is a stand-alone Common Lisp program, and can analyze data from any source.

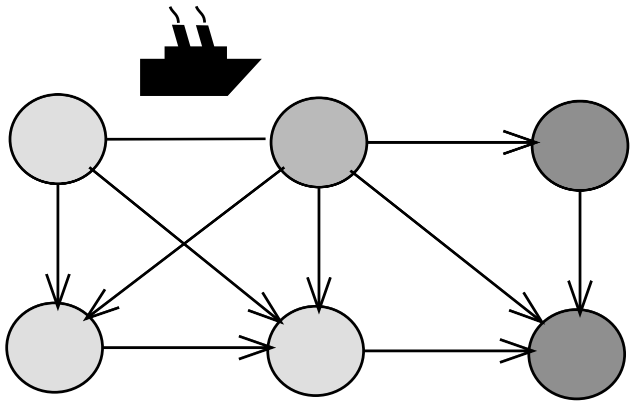


Figure 1: TransSim simulates ships moving cargo between ports. The shade of gray in the port represents how full it is; dark ports are near capacity. In the real interface, the colors green, amber and red are used rather than gray levels.

researcher will want to analyze the data using statistical software. While the data files that Clip writes can be analyzed by any statistical package, Clip is especially well integrated with Clasp, which is a statistical package that EKSL has implemented. Clasp has many of the standard descriptive and inferential statistics, together with a convenient graphical user interface, and a Lisp interaction window that researchers can use for implementing statistical operations that we have not anticipated.

## 2 Transportation simulator

TransSim simulates the execution of transportation plans in a problem domain where the goal is to get cargo through a shipping network from a number of starting locations to a number of destination ports. The problem, defined as part of the Arpa/rl[2] Planning Initiative, involves many different kinds of cargo and ships, and many, many pieces of cargo. TransSim allows the user to configure an arbitrary shipping network by specifying ports, a set of cargo inputs, and a list of available ships. Ports and docks can have constraints on the kinds of ships and cargo they can handle. Input to the scenario is a list of Simple Movement Requirements (SMRs). SMRs specify a port of embarkation, various intermediate ports, and a port of debarkation. Cargo appears at its port of embarkation at times determined by the scenario and travels through the network along the route specified by its SMR. The time for a ship to travel between ports is a Gaussian random variable computed by the simulator and controlled by user-specifiable parameters giving the mean and variance of the ship's speed. Figure 1 illustrates the basic idea of TransSim. The links in the figure only show the connectivity of ports, not the actual distances between them.

TransSim also supports Interactive Plan Steering, where a human user or a software agent notices problems (pathologies) in the execution of a plan and intervenes in an attempt to get the plan back on course. Currently, our Plan Steering Agent works without reference to a plan or schedule. It attempts to control the shipping traffic by using limited look-ahead for prediction, and it reacts to pathologies as they are detected. One important kind of pathology occurs when the number of ships arriving at a port exceeds the capacity of the port (the number of docks), so that the ships must wait until docks are available before they can unload. For example, in figure 1, the Plan Steering Agent might try to re-route cargo traveling through the port at the upper right, since it is near to being a bottleneck.

We have developed a "pathology demon" to try to predict this pathology. Its prediction is for a specified number of days in the future, say four days. The demon looks at each ship heading for a particular port and uses the mean and variance on the ship's speed to estimate the probability that the ship will have arrived by the day in question. If that probability exceeds some threshold, the demon assumes that the ship arrives.

[2] Advanced Research Projects Agency and Rome Laboratory

The demon also predicts how many ships will leave the port by that day, using heuristic estimates about the time it takes to unload and load a ship. All this information is compiled into an estimate of how many ships will be in port on the day in question. If the estimate is higher than the port capacity, the pathology demon can alert the Plan Steering Agent (who may be human); the Plan Steering Agent can then decide what to do, which might include re-routing some of the ships. On the other hand, the Plan Steering Agent might ignore the problem because of global considerations. The pathology, of course, is only a local problem, and may be no great hindrance to the overall plan. To study the extent to which local problems affect plan performance, or whether the pathology demon is good at predicting the number of ships in port, or any of myriad other questions, we will need to run experiments, collect statistics, and analyze them. To do that, we will use CLIP and CLASP.

## 3 Running experiments

A great many experiment designs are used in science, but most of them can be viewed as sets of ***trials***, each with a number of independent variables, representing the conditions under which the trial is run, and a number of dependent variables, which are the objects of scientific scrutiny. This is the simplest of the kinds of experiment designs that CLIP supports.

One common kind of experiment within this paradigm is called a "fully factorial" design, in which there are one or more ***factors***, each of which has a small number of discrete levels. For example, factor A might be the number of days in advance that the pathology demon tries to predict the number of ships in port, with three values (levels)—2, 4 and 6 days. Factor B might be the probability threshold, above which the demon assumes that the ship will be in port, say with levels 0.25, 0.5 and 0.75. A fully factorial experiment design will test all combinations of levels; in this example, there are nine experiment conditions. Because of random variation in the outcome of each trial, the experimenter will usually want multiple trials in each condition and will probably analyze the data using the statistical technique of analysis of variance. It's easy to do this kind of experiment using CLIP and CLASP: we tell CLIP how to modify the parameters of the pathology demon and it takes care of iterating through all the conditions, setting the parameters, and collecting the data. Later, CLASP can analyze the data, using just a few mouse clicks, since the analysis of variance is built in.

Another common kind of experiment looks at the relationship of two or more continuous variables, such as the correlation between them. For example, the independent variables (variables controlled by the experimenter) might be the number of cargo units to be shipped and the amount of variance in ship speed, while the dependent variable (a variable measured by the experimenter) might be the amount that the plan is late or the number of missed deadlines. We expect that as the scenario becomes more difficult (when the values of the independent variables increase), the plan lateness and missed deadlines will go up—but will this relationship be linear or non-linear? To answer such questions, we will want to run many trials, choosing values for the continuous independent variables and measuring the dependent variables. CLIP can help us do this, while CLASP can graphically display the data and transformations of it, together with regression lines, if desired.

### 3.1 Instrumentation

Adding code to extract information from a system is called ***instrumentation***, hence CLIP's name (Common Lisp Instrumentation Package). Most of CLIP's functionality is directed towards extracting different kinds of information from the target system—information that is calculated afterwards, collected periodically during execution, or collected whenever some interesting event occurs. This aspect of CLIP is deferred to section 3.2. First, we present an overview of how CLIP works and what you need to do to use it. (This article is no substitute for the CLIP/CLASP manual [2], where everything is rigorously explained.)

To use CLIP to run an experiment, CLIP first needs to know how to run your simulator (or whatever your program is). Essentially, this is a single function or piece of code that CLIP can call to start a trial and which will return when the trial is over. CLIP also works with simulators that run in multi-threaded (multiple process) Lisps, but it nevertheless treats the simulator as a single piece of code.[3] Between trials, CLIP will

[3] This requirement may be lifted in future versions of CLIP, but the impact is minor. Most multi-threaded Lisps provide a

```
(define-simulator transsim
  :system-name "TransSim"
  :start-system (simulate nil)
  :reset-system reset-transsim-experiment)
```

Figure 2: An example of the **define-simulator** macro.

```
(define-experiment test-experiment ()
  :simulator transsim
  :instrumentation (prediction-score port-state-snapshot)
  :variables ((prediction-threshold in '(0.1 0.2))
              (eta-variance-multiplier in '(0.05 0.15 0.25))
              (prediction-point in '(2 4 6)))
  :before-trial
    (setf *ports-to-consider* (list (port 'port-1)))
  :after-trial
    (write-current-experiment-data))
```

Figure 3: An example of the **define-experiment** macro.

need to reset your system, although this might be unnecessary if the simulator is purely functional (few are). If your simulator has a notion of time, such as having a clock, and you want CLIP to schedule events for particular times, CLIP will need to know how to interact with the scheduler and the clock. For example, you might want to collect data every day of the simulation, with the average being written to the data file. To describe how to run and control your simulator, there is a single CLIP macro, called **define-simulator**.

Figure 2 shows a very simple example of the macro. (A more complete example appears in figure 5.) Notice that the macro tells CLIP how to run TRANSSIM, namely to call **simulate** with an argument of `NIL`, and similarly the macro tells how to reset the simulator between trials and so forth.

Next, you will define your experiment, which is again done with a single CLIP macro, called **define-experiment**. The heart of an experiment is the set of independent and dependent variables, which are specified using that macro. The independent variables are described with a simple syntax much like the Common Lisp **loop** macro. The names of dependent variables are simply listed—how to collect and report the data for each dependent variable is separately defined via objects called "clips," which will be discussed in the next subsection.

Figure 3 shows a simple example of the **define-experiment** macro. Notice how the macro refers to the TRANSSIM simulator, telling CLIP how to run the user's program. Next, it mentions the names of the data to collect; later we will see how those are defined. The **:variables** are the independent variables—this experiment has three, with 2, 3 and 3 levels, for a total of 18 conditions.

The **define-experiment** macro also provides ways for users to run code of their own choosing during the experiment, at four distinct times. Those times correspond roughly to the following pseudo-code definition for an experiment:

```
before experiment
LOOP
    before trial
    RUN TRIAL
    after trial
ENDLOOP
after experiment
```

`process-wait` function, which can be used to make the simulator seem like a single piece of code.

These opportunities to run code might be used as follows:

**Before the Experiment:** When an experiment gets started, you may want, for example, to load special knowledge-bases or set scenario parameters. This is also a chance to do more mundane things, such as allocating data structures or turning off the screen-saver (some computer systems get confused when writing to a screen that is under the control of a screen-saver).

**Before Each Trial:** At each trial, you may want to reinitialize parameters and data structures. One important thing to do is to configure your simulator for the current experimental condition. For example, if you are running a two-factor experiment, CLIP will have two local variables bound to the correct values of those two factors. You may then use those variables to, perhaps, set parameters of your simulator or use them as arguments to initialization functions. After all, only you know the semantics of your factors.

**After Each Trial:** The most important thing that is typically done after each trial is to call the function **`write-current-experiment-data`**, the CLIP function that writes all the data for this trial. (You can see this in the **`define-experiment`** example above.) This is also a good time to run the garbage collector, if you want to minimize garbage collection during trials.

**After the Experiment:** Typically, code run after the experiment undoes the code run before the experiment, such as deleting data structures or turning the screen-saver back on.

Of course, any arbitrary code can be executed at these times, for whatever purposes you want. The key idea is that the before- and after-trial code surrounds every trial and runs many times, while the before- and after-experiment code surrounds the whole experiment and runs only once. This ability to run arbitrary code is more than just an opportunity for hacks—it is a clear and precise record of the exact experimental conditions. Records are important as a memory aid and as a means for replicating experiments.

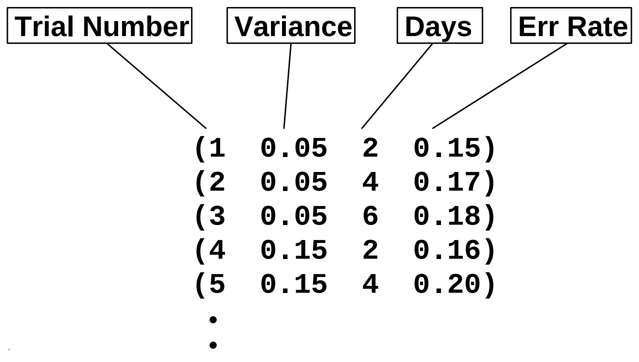
When the experiment has been defined, you start it running with the function **`run-experiment`**. This function takes arguments, which you can refer to in the before/after code, so that the final specification of the experimental conditions can be deferred until run-time. (For the sake of record-keeping, these arguments should be written to the data file, by using the CLIP function **`append-extra-header`** in the before-experiment code.) The **`run-experiment`** function also allows you to specify the output file for the data, the number of trials, the length of the trial, and other such information. An example appears in section 5.

Defining the simulator and the experiment, and then running the experiment is fairly straightforward and is only a fraction of what must be done to run an experiment. The bulk of the effort is in defining "clips"—functions that measure the dependent variables of your experiment. Fortunately, they are modular and reusable.

## 3.2 Clips

Clips are named by analogy with the "alligator clips" that connect diagnostic meters to electrical devices. They measure and record aspects of your system (the values need not be numerical). Essentially, they are Lisp functions that you define and which CLIP runs if they are included in the definition of the experiment. Once written, they can be mentioned in any number of experiments. Indeed, it's common to build up files of clips, so that a new experiment can be quickly defined by writing a **`define-experiment`** form (or editing an old one) and listing the clips in the **`:instrumentation`** argument to **`define-experiment`**.

Clips are defined with the **`defclip`** macro, which is syntactically very much like **`defun`**, except that information given before the function body is read by CLIP. The function body is entirely up to you, the user; its purpose is to extract information about the state of your system. What CLIP needs to know is the time that it should run the clip. Most clips simply measure values after a trial is finished, for variables such as "finish date," "number of bottlenecks," and "total waiting time for ships." More complicated clips may need to run periodically, although this only makes sense for simulators that have a clock of some sort. Time-dependent clips are scheduled using the **`schedule-function`** specified in the **`define-simulator`** form. (This aspect of the macro is illustrated in the example in figure 5.) Other clips may need to run when some particular event happens; this is accomplished by tying the clip to a function in your simulator that is associated with that event, using a mechanism like the "advise" facility found in many Lisp implementations.



```
(1  0.05  2  0.15)
(2  0.05  4  0.17)
(3  0.05  6  0.18)
(4  0.15  2  0.16)
(5  0.15  4  0.20)
 •
 •
```

Figure 4: Clip writes data to a file in a format like this. Each row corresponds to the data for a single trial. Generally, each clip function writes a single element of the row, although clips that write multiple elements are easily defined.

Consequently, the **`defclip`** form has syntax for tying the clip to a user function. When a clip is run many times during a trial, it can either report the mean of the values or it can report all the values (or some function of them), as ***time series*** data (see section 3.3). Figure 4 shows the format of a Clip data file.

Clip implements several features to make clips more useful and powerful. The first feature allows a clip to report several values to the data file. In other words, if we think of the data file as a large table (see figure 4), with a row for every trial and a column for each variable, a clip may report the values for several columns. For example, a clip that interrogates a port might want to report the minimum, maximum, and mean queue length. The user can define a clip called **`queue-info`** to report all three of these values during an experiment. The second feature allows users to report a value for each of several objects. For example, they might want to report the maximum queue length at each port, or the tons of cargo carried by each ship. Given a clip to report the value for a single object, another clip can be defined that maps over the objects, calling the simpler clip for each object. These two features can be combined, yielding one clip that reports a lot of information about many objects, all in one powerful step. An important restriction is that the number of values must be consistent, because the data files need to have the same number of values (columns) reported for every trial. This is not a requirement of Clip so much as a requirement of the statistical package, whether Clasp or any other package. Missing values are a headache for any statistical operation, and so it is better to always produce the same number of values. Typically, this is easy to accomplish. For example, the number of ports should be the same in every trial. If they are not, you will probably be comparing average behavior (since you cannot compare them pairwise), in which case the average can be reported, rather than data for each port. Section 5 describes an entire experiment, showing how all the pieces described in the last few sections fit together.

## 3.3 Time series data

So far, we have described different kinds of data that can be extracted into a “snapshot” of the scenario. We can also collect data that is a “movie” of the system: a series of snapshots at different points in time. Data like this is called ***time series*** data. For example, we could report the queue length at a port each day, allowing us to see bottlenecks arise and subside as the traffic ebbs and flows. We can statistically analyze such data to see if there are temporal correlations. For example, we could see whether a bottleneck truly subsides or merely moves to another port at a later time. We just cannot answer such questions by looking at mean values after a trial is over.

Early in our work with TransSim and the pathology demons, we wanted to see whether the prediction accuracy (error rate) of the pathology demon varied over time. The demon might, for instance, become increasingly confused as time progresses. Therefore, we need a clip to run every simulated day, to record a large amount of data about the state of ports. Two clips are shown in figure 5. (The figure also shows the more elaborate definition of the simulator that is needed to tell Clip how to schedule data collection

```
(define-simulator transsim
  :system-name "TransSim"
  :start-system (simulate nil)
  :reset-system reset-transsim-experiment
  ;; a function that places functions to run on the queue of events.
  :schedule-function schedule-function-for-clips
  ;; a function that removes functions from the queue of events.
  :deactivate-scheduled-function transsim::reset
  :seconds-per-time-unit 3600
  :timestamp current-day)

(defclip port-state-snapshot ()
  "Record state information for a port at the end of each day."
  (:output-file "christa:oates.data;port-time-series.clasp"
   :schedule (:period "1 day")
   :map-function *ports-to-consider*
   :columns (ships-en-route
             ships-queued
             ships-docked
             expected-ship-arrivals
             predicted-queue-length
             time-to-clear)))

(defclip ships-en-route (port)
  "Record the number of ships en route to a port."
  ()
  (length (apply #'append (mapcar 'contents (incoming-channels port)))))
```

Figure 5: A simulator definition and time-series clips to capture the state of the ports during a TransSim simulation.

at particular times; compare this to figure 2.) The first clip is one of the complex clips that were described earlier: it maps over a set of ports (`*ports-to-consider*`) and collects a bunch of information on each one, as specified by the `:columns` keyword argument. Each column entry is the name of a clip that simply reports one value. One example is the `ships-en-route` clip, which is also shown in the figure. Finally, of course, we see that `port-state-snapshot` is scheduled to run once a day.

You may have also noted that `port-state-snapshot` specifies its own output file. The reason is that time series data is incompatible with the data collected after the trial. Different kinds of data are collected by time series clips: individual values during a trial versus means and totals afterwards. Usually, a different number of values are produced. It doesn't make sense to mix the two. Therefore, time series data are written to a different file than the main data file. In fact, you can collect several different kinds of time series data in one experiment. For example, you can collect information on port queues every day and collect information every time a ship is loaded. Again because of incompatibility of the data, these two different time series would be written to different files. Someone with such a complex experiment often makes a directory into which all of the data files will go.

## 3.4 Summary

The capabilities of Clip have been driven by the needs of experimenters. There are a great many features, all of which have proven useful to someone. Nevertheless, the essence is fairly straightforward. To run an experiment using Clip, you must do the following: (1) define the simulator, (2) define the clips, (3) define the experiment, and (4) run the experiment.

CLIP has other features to support experimentation, such as aborting a trial but continuing the experiment, say when some intermittent error has occurred—very common in stochastic simulations. CLIP also lets you run only part of the experiment, which facilitates breaking the experiment into parts to run on different machines. These are all explained at length in the CLIP/CLASP documentation [2].

## 4 Data Analysis

The idea of CLASP began when we wanted to run a $t$-test on some experiment data without having to write out the data to a file in some tab-delimited format, move the code to another machine, run a statistics program, and load the data. From this small beginning, we have added most of the workhorse statistical functions, data manipulation (regrouping, selecting subsets), data transformation (such as log transforms), and graphing software (now replaced by SCIGRAPH, by Bolt, Beranek and Newman, Inc.).[4] We have a convenient graphical user interface implemented in CLIM, and a programmatic interface so that CLASP functions can be called by the user if a desired data manipulation isn't already on a menu. Ideally, everything can be accomplished by menus in the graphical user interface.

CLASP's screen interface, an example of which is shown later in figure 6, comprises four areas: the menus, the datasets, the results, and the notebook:

**Menus** The CLASP menus will appear across the top of the window. See, for example, figure 6. The menus, which will be discussed below, are: File, Graph, Describe, Manipulate, Transform, Test, and Sample.

**Datasets** When you load a file of data into CLASP, such as a file written by CLIP, it becomes a CLASP *dataset* and appears on this menu (the upper left of figure 6). The name of the dataset is the name of the experiment. Each column of data is called a *variable*; the name of the variable is usually the name of the clip that returned that variable, unless you specify a different name in the **`defclip`**.

When analyzing the main data file (as opposed to a file of time series data), there will be as many variables as there were clip values, and each variable will have as many elements as there were trials, since each clip reports once at the end of each trial. (CLIP has a naming scheme to handle clips that produce multiple values.)

Most operations in CLASP take either datasets or variables as arguments, and the items in the dataset pane become mouse-sensitive when appropriate. For example, if you want to find the mean number of days cargo spends in transit (and you had a clip that reported that value), you would just select the "Mean" item from the "Describe" menu, whereupon all the variables would become mouse sensitive, and you could select the one you want. Similarly, when you want to partition a dataset, say to separate trials where the Plan Steering Agent was used from those where it wasn't, you would first select the "partition" command from the "Manipulate" menu item, and then click on your dataset.

**Results Display** When a CLASP operation yields a complex result, such as a table or graph, the name of that object goes into a menu of results (the lower right of figure 6). The most common use for this menu is to bring up two results side-by-side, so they can be compared. Graphs can often be overlaid, so that similarities are obvious. There are also CLASP commands to delete, print, display, and otherwise operate on results, whereupon they become mouse sensitive.

**Notebook** The notebook is by far the largest part of the CLASP window because most of the action goes on here (in figure 6, it is the large pane overlaid with the graph). It is a complete Lisp read-eval-print loop, except that CLASP commands are also accepted. Having Lisp available is important and powerful, because users can operate on the data in ways we have not yet implemented or even thought of.

CLASP commands can be typed instead of using the menus; indeed the menus just type the appropriate thing into the notebook. When the command is fully entered, it's executed and its results are printed to the notebook. CLASP output in the notebook is also mouse-sensitive when appropriate.

One of the nice features of the notebook is that it provides a record of the statistical operations on the data. This record can be saved to a POSTSCRIPT® file and printed.

[4] CLIP and CLASP have been integrated into the Common Prototyping Environment of the ARPA/RL Planning Initiative as a tool for evaluating planning technology. See section 7.

CLASP uses a prefix command syntax, very much like Lisp, in that you give the command name first, such as **:T Test Two Samples** *X Y*, where *X* and *Y* are variables. Using the features of CLIM, CLASP allows command completion and prompts for arguments. CLASP also allows certain arguments to be "mapped," which means that when a list of arguments is given where a single argument is expected, the command is executed for each element of that list. For example, to find the means of three variables, (**X Y Z**), you can use the following syntax:

```
:Mean X,Y,Z
```

CLASP groups related commands in the main menus, as sketched below; full information is in the CLIP/CLASP manual.

**File** This menu allows you to load CLASP datasets from files and to save them to files, say if you've made changes or created new datasets. It also allows you to read and write datasets in formats understood by other statistical packages. A number of other utilities are on this menu, such as printing objects (graphs or tables) to POSTSCRIPT files.

**Graph** Being able to look at your data in various ways is important in exploratory analysis. You may find discontinuous or skewed distributions, non-linearities in trend, or peculiar clusters of data. Looking at the data will suggest new hypotheses and statistical operations, such as smoothing or correlation. This menu allows a number of displays of data, including histograms, scatter plots, line plots, and regression plots. The grapher, BBN's SCIGRAPH, allows graphs to be overlaid for ease of comparison. It also allows the objects (points or lines) in a plot to be colored based on some other property, another important tool for exploratory data analysis.

**Describe** Statistics are often divided into descriptive statistics and inferential statistics. The former are functions that capture some property of one or more samples, such as location (mean, median), spread (variance, interquartile range) or other properties (correlation between two variables). The latter are functions that test hypotheses about the populations that the samples were drawn from. This menu contains many of the descriptive statistics, including all the ones just mentioned, and a few others, such as modes, trimmed means, arbitrary quantiles, cross-correlations, and auto-correlations. There is also a "statistical summary" operation that prints most of the interesting one-sample statistics in one convenient table.

**Manipulate** An experiment usually produces lots of data, which must be broken into pieces to be looked at and understood. Therefore, CLASP provides several ways to extract subsets from a dataset. One example is partitioning, where you select a dataset and a categorical variable from that data. A categorical variable has a few discrete values: for instance, in the shipping domain used in TRANSSIM, the variable **ship-type** might have discrete values like **container**, **tanker**, and **Roll-on/Roll-off**. The partition operation produces new datasets (which appear in the dataset window), one for each distinct value of the categorical variable. You can then select one of these datasets if you want to look just at one value of the variable, say, the "Roll-on/Roll-off" data. Similar operations allow you to partition datasets by an arbitrary predicate (one that you type in).

Other operations on this menu allow you to create new datasets. The values for these new datasets may be cobbled together from existing datasets or come from Lisp functions you execute in the notebook.

When new datasets are produced, whether by partitioning or other operations, a new name is generated, by combining the old name with the operation. This means you can often remember what a dataset is just by looking at its generated name. For example, a dataset **SHIPS** that has been partitioned by its **TYPE** variable, which has a **TANKER** value, will result in a new dataset named **SHIPS (TYPE = TANKER)**.

**Transform** This menu has commands that produce new variables from old ones. A trivial example is just to sort the variable. A more interesting one is a logarithmic transformation that might be used prior to linear regression, resulting in an exponential model of the data. Another example is smoothing the data, which might be used prior to autocorrelation in order to find cyclical patterns in time-series data. As with datasets, when a new variable is produced, a new name is generated by combining the old name with the operation. For example, a variable named **QUEUE-LENGTH** that has been smoothed will result in a new variable named **SMOOTH-OF-QUEUE-LENGTH**.

**Test** This menu contains the inferential statistics that were omitted from the Describe menu. Most of these commands, such as the *t*-test, confidence intervals, analysis of variance, chi-square and regression, are described in any statistics textbook.

**Sample** This menu contains commands that produce artificial data by sampling from a given probability distribution. These commands would rarely be used in ordinary data analysis, but they are pedagogically useful to see how various graphing options and statistical tests work on data with known properties. The commands can draw numbers from the uniform, normal, binomial, Poisson, and gamma distributions.

## 5 Example

Rather than try to describe in detail how Clip and Clasp work, we will present an example in which we use them to run and analyze an experiment in the Transportation Planning domain. The example uses the TransSim simulator and is based on a pilot experiment that Tim Oates used in designing his Plan Steering Agent [3, 4]. The purpose of the experiment is to assess the error rate of a demon that predicts the queue length at a port *d* days in advance, as a function of the variability of ship speed and the time delay, *d*.

The following defines the TransSim simulator. It's quite simple because we won't be using any time-series collection in this experiment. A more complete example was given earlier, including the information necessary for time-series data.

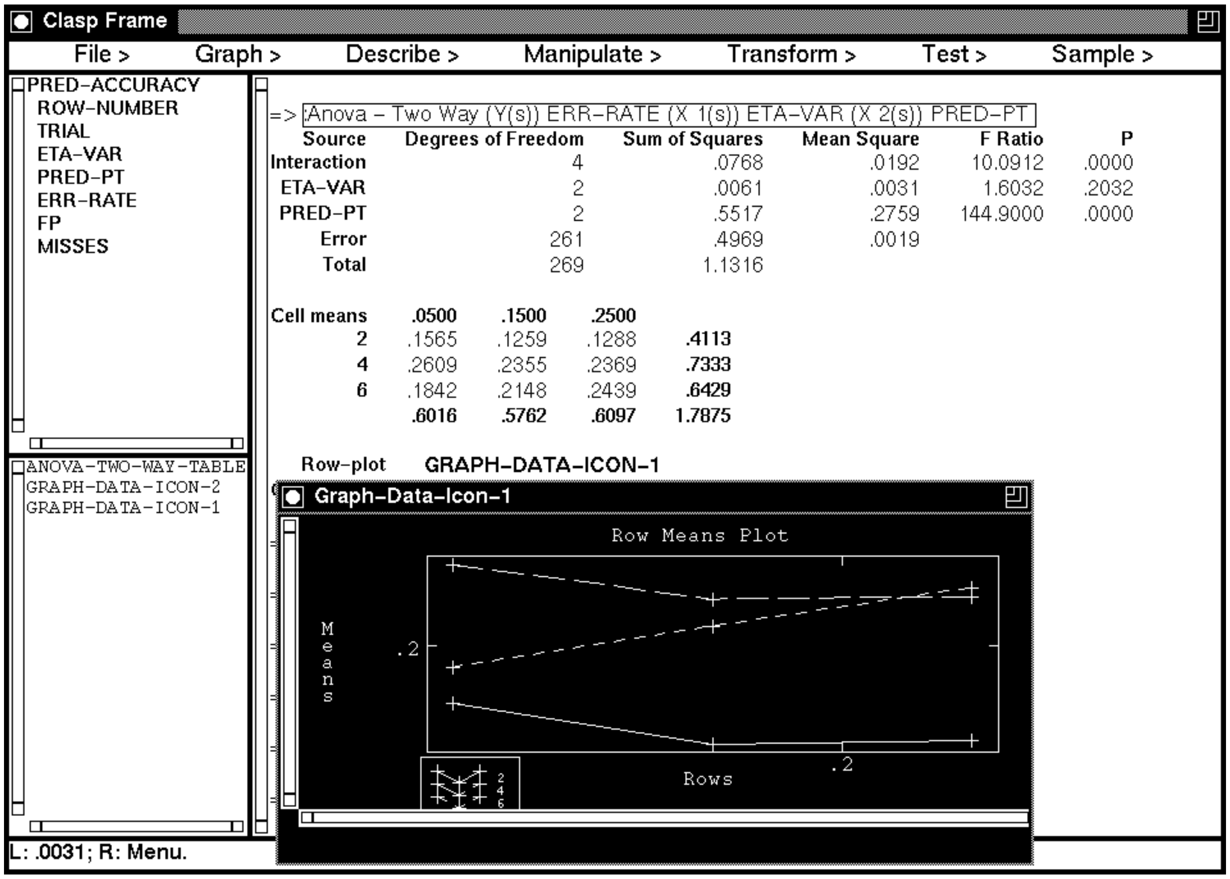
```
(clip:define-simulator transsim
  :system-name  "TransSim"
  :start-system (simulate nil)
  :reset-system initialize-simulation)
```

Our example experiment will measure the accuracy of the demon that predicts queue lengths at ports and is defined below. Its `:instrumentation` clause mentions three clips for the dependent variables: in this experiment, we are interested in the error rate of the demons in predicting queue length, and in their misses and false positives in predicting bottlenecks. The `:variables` clause specifies two independent variables—the variance in ship arrival time and the number of days in the future to predict the queue length. In this experiment, the only thing to do before each trial is to transfer (using `setq`) the values of the independent variables to the appropriate global variables of the TransSim simulator. After each trial, the trial number and the values of independent variables and the clips are written to the data file.

```
(clip:define-experiment pred-accuracy ()
  :simulator         transsim
  :instrumentation   (err-rate fp misses)
  :variables         ((eta-var in '(0.05 0.15 0.25))
                      (pred-pt in '(2 4 6)))
  :before-trial      (setf *eta-variance-multiplier*
                           eta-var
                          *prediction-points*
                          (list pred-pt))
  :after-trial       (write-current-experiment-data))
```

Below is the code for one of the clips in the experiment. It looks just like a Lisp `defun`, except for the `()` before the code. That list is used for specifying additional information such as whether this is a time series clip (by default, clips are not time series), whether it maps over several objects, and so forth. The information is specified in keyword style, as was shown in the example in figure 5. Since each port has its own prediction demon, this clip reports the mean error rate over all the demons.

```
(defclip err-rate () ()
  (loop for p in *ports*
        sum (demon-error-rate (port-demon p))
            into total
        finally (return (/ total (length *ports*)))))
```



```
=> :Anova - Two Way (Y(s)) ERR-RATE (X 1(s)) ETA-VAR (X 2(s)) PRED-PT
```

| Source | Degrees of Freedom | Sum of Squares | Mean Square | F Ratio | P |
|---|---|---|---|---|---|
| Interaction | 4 | .0768 | .0192 | 10.0912 | .0000 |
| ETA-VAR | 2 | .0061 | .0031 | 1.6032 | .2032 |
| PRED-PT | 2 | .5517 | .2759 | 144.9000 | .0000 |
| Error | 261 | .4969 | .0019 | | |
| Total | 269 | 1.1316 | | | |

| Cell means | .0500 | .1500 | .2500 | |
|---|---|---|---|---|
| 2 | .1565 | .1259 | .1288 | .4113 |
| 4 | .2609 | .2355 | .2369 | .7333 |
| 6 | .1842 | .2148 | .2439 | .6429 |
| | .6016 | .5762 | .6097 | 1.7875 |

Row-plot GRAPH-DATA-ICON-1

Figure 6: Excerpt from sample interaction with CLASP.

The experiment is run by executing the following form. The **:repetitions** clause says how many trials to run under each condition (combination of levels of the independent variables). In this experiment there are nine conditions (3 levels of variance and three prediction points), so with thirty repetitions in each, CLIP will run 270 trials, resulting in 270 rows in the output file.

```
(run-experiment 'pred-accuracy
    :output-file "~/data/demon-summary.clasp"
    :repetitions 30)
```

When the experiment is complete, we will want to analyze the data using CLASP. We are interested in whether either independent variable affects the demon's error rate, and, if so, whether those effects interact. Therefore, we will analyze the data with a two-way analysis of variance (Anova). Obviously, we cannot show the sequence of mouse-clicks that did the analysis, but Figure 6 shows the CLASP screen afterwards. The data show that there is a significant interaction between the two factors ($F = 10.09, p = 0.0$), because increasing variance didn't affect the error rate much when predicting two and four days in advance, but greatly increased the error rate when predicting six days in advance. We have superimposed a CLASP-generated graph to depict the interaction; note that one of the lines slopes upward, while the other two decline slightly. The data also show that, overall, the point of prediction was highly significant ($F = 144.9, p = 0.0$), but the amount of variance in the ship speed was not ($F = 1.6, p = 0.2$).

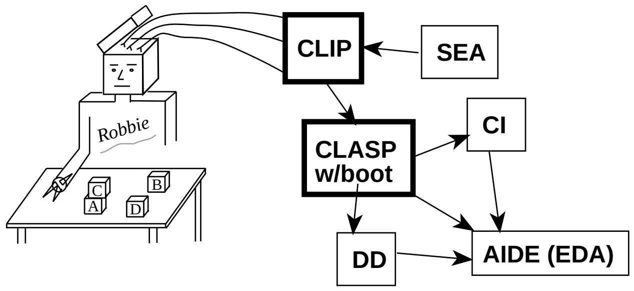


Figure 7: Additional experiment modules are being added to CLIP and CLASP. The SEA system will help design experiments using CLIP. CLASP has been augmented with bootstrapping functions, and provides basic statistical operations for Dependency Detection (DD), Causal Induction (CI), and automated data analysis, including AIDE and other Experimental Data Analysis work.

## 6 Empirical Analysis Toolbox

CLIP and CLASP were originally developed to analyze the behavior of an AI planning system *online*—that is, to instrument the system, run experiments, analyze the results, and build predictive models of the planner's behavior, all in the same Common Lisp environment. We were successful in the sense that CLIP and CLASP are well-integrated and provide the simple tools we need to model behavior. However, we discovered that using this simple tool set has two problems: tedium and lack of expertise. It can be tedious and time-consuming to use basic instrumentation and statistics methods to address sophisticated modeling issues. (For example, what are the important contributing factors to plan failure and do their relative influences change as environmental conditions worsen?) Moreover, while tedious, using such methods requires care and some experimental and statistical sophistication to produce sound models.

Commercial statistical packages face the same problem, and like many, we have chosen a solution that provides add-on modules that work with CLIP and CLASP to perform specific kinds of data analysis and modeling tasks. Unlike other packages, our modules are designed specifically for modeling the behavior of AI programs. Each is tailored to a specific aspect of program analysis, such as finding the major factors contributing to program success or identifying interactions of program components that degrade performance. The modules we are currently building, while by no means complete, include four that have proven particularly useful and are described below. Also unlike other packages, our add-on modules incorporate AI techniques wherever possible (knowledge representations, search heuristics, reasoning strategies and so forth), in the hope that we can eventually automate many aspects of empirical analysis that are now done under the user's direction.

**Exploratory Data Analysis** After running an exploratory experiment and gathering data, the user is faced with the task of identifying significant relationships among the factors measured. This is called Exploratory Data Analysis (EDA). We are building a module that assists the user in this effort by employing EDA techniques [5,6]. These techniques can partition data to distinguish different modes of behavior and generate functional descriptions of interactions between factors. Through detailed exploration of experimental data the user can gain a more complete picture of program behavior.

**Bootstrapping** Bootstrap statistics [7,8] replace the parametric and distributional assumptions of statistics like the $t$ test with an empirical approach using computerized resampling of the data. This frees the user from many of the restrictive parametric assumptions made by commonly used statistics. The $d$ test, for example, is used just like the two-sample $t$ test, especially when the data don't satisfy the normality and equal-variance assumptions of the $t$ test.

We have implemented a number of resampling techniques including Monte Carlo and bootstrapping. These are generic techniques that can be used with most kinds of statistical functions. The idea is to specify an arbitrary univariate or bivariate statistical function (such as the median or correlation coefficient), one or two sets of data, and the number of times to resample. Random samples are drawn from the data,[5] the function is run on the random sample, and the result recorded. The distribution of the results can indicate what the distribution of the statistic is under the null hypothesis, and hence the significance of the statistic when computed on the actual data. CLASP's bootstrapping code can work with any statistical function, including one defined by the user. A *d* test, described above, could have been specified as follows:

```
(compute-bootstrap-special #'mean-diff sample-1 sample-2 1000
                           :mode :joint-with-replacement)
```

This would run a user-defined `mean-diff` function on each of 1000 pairs of samples, where `:joint-with-replacement` means that the elements of each sample are drawn randomly from the union of *sample1* and *sample2*.[6] The `mean-diff` function computes the difference in the means of two samples. The distribution of these 1000 values indicates how significant the true difference of the means is, just as in the *t* test.

**Dependency Detection** The complexity of AI programs has grown to a point where their behavior is difficult to predict and problems difficult to replicate. Program actions often interact in unforeseen and deleterious ways. We employ a technique we call ***dependency detection***, analyzing program execution traces with a statistical filter to find significant dependencies among interacting actions [9]. Once identified, these dependencies can be examined more carefully to find and fix the unforeseen interactions that often cause them. We have successfully employed dependency detection to identify and debug such interactions in the PHOENIX planner, using execution traces that consist of a single stream of tokens representing each action taken by the planner (including plan repair actions) and each recorded instance of plan failure.

This single-stream form of dependency detection has recently been enhanced by the incorporation of efficient heuristic techniques to guide the search for dependencies [10]. In addition, we have developed an algorithm for ***multi-stream dependency detection***, which identifies significant dependencies in multiple concurrent streams (for example, the various sensor readings taken for an intensive care patient). Using this algorithm, we generated a set of predictive rules for port bottlenecks in TRANSSIM that were better than the hand-crafted rules previously used [11].

**Causal Induction** Other techniques, in addition to dependency detection, can be employed to model program behavior. A predictive model should tell us how we can change the program to improve or correct its behavior. This requires that we understand the underlying causal relationships among the factors influencing its behavior. We are developing a module that builds causal models from data [12]. The input to this module is a set of factors whose relationships are to be modeled. The output is a graph with factors at the nodes and with arcs that show both causal direction and strength of influence.

Some of the causal induction techniques are based on ***path analysis*** [13, 14]. In addition, this module will provide several new algorithms that induce structural equation models from data [15]. These algorithms, which are based on linear regression, compare quite favorably with two other well-known causal modeling algorithms that are based on conditional independence: the **IC** algorithm by Pearl and Verma [16] and the **PC** algorithm by Spirtes, et al. [17], often outperforming them. This is most remarkable given the relative simplicity and computational efficiency of these algorithms when compared to **IC** and **PC**. The key is our reliance on regression techniques [18] and on a simple filtering heuristic we call $\omega$. (Roughly, $\omega$ is the percentage of the correlation that is indirect.)

Case studies using these and other empirical techniques to analyze AI programs are included in a forthcoming textbook on empirical methods for AI research [19]. It is possible that the major contribution of CLIP/CLASP will not be as a standalone instrumentation and analysis package, but rather as a platform for the integration of more powerful techniques such as those described above. We envision a new generation

[5] Various resampling schemes are possible, such as whether the elements are drawn with replacement and whether bivariate data can be exchanged between groups.

[6] Actually, the concatenation of the two samples, in case there are duplicates.

of statistical software in which knowledge and heuristics will guide application of the data analysis and modeling techniques described above.

With this vision in mind we are currently developing the Assistant for Intelligent Data Exploration (AIDE) to assist human analysts in exploratory data analysis (EDA) [5]. AIDE adopts a planning approach to automating EDA. Data-directed mechanisms extract simple observations and suggestive indications from the data. Scripted combinations of EDA operations are then applied in a goal-directed fashion to generate simpler, deeper, or extended descriptions of the data. The system is mixed-initiative, capable of autonomously pursuing high- and low-level goals while still allowing the user to guide or override its decisions. It is also modular and has already incorporated several of the causal-modeling algorithms described above to drive goal-directed processing.

Another intriguing problem that is a candidate for partial or full automation is experiment design. We are currently designing a Scientist's Empirical Assistant, SEA, that will provide an intelligent, goal-driven approach to creating scientific experiments [20]. SEA will use a large repository of knowledge to create one or more ***experiment plans***. CLIP will provide the basic building blocks by which experiments are run. We are developing a representation based on ***function modeling*** to model the knowledge and information flow required to design empirical experiments such as those we use to analyze AI programs.

## 7 Related Work

We know of no other Common Lisp instrumentation tool like CLIP, with the exception of a package called METERS by Bolt, Beranek and Newman, Inc. [21]. Like CLIP, METERS was developed for use in the ARPA/RL Planning Initiative (ARPI). Unlike CLIP which primarily (though not exclusively) focuses on instrumenting a single AI program, METERS is designed to measure the performance of various AI programs running in a distributed network called the Common Prototyping Environment, or CPE. AI components such as planners, case-based reasoners and schedulers, often running on different machines, work together in the CPE on different aspects of a common problem. This environment allows ARPI developers to mix and match different components (for example, swapping one planner for another) to test their relative performance within the larger system. METERS provides a number of helpful facilities for profiling components in such a distributed system, such as the time spent inside individual modules or the communication overhead for each module and among subsets of modules.

Two well-developed Common Lisp statistical packages provide alternatives for the user requiring statistical or quantitative reasoning methods not (yet) implemented in CLASP: XLISP-STAT and QUAIL.

XLISP-STAT [22, 23] was developed under Luke Tierney at the School of Statistics at the University of Minnesota. XLISP-STAT provides a rich set of statistical and dynamic graphing capabilities. Some features found in XLISP-STAT and not in CLASP include: nonlinear regression, maximization and minimum likelihood estimation, and approximate Bayesian computation. Written in a subset of Common Lisp (though not including CLOS), XLISP-STAT provides many extensions to that subset to support further development of statistical methods. Originally developed for the Apple Macintosh,™ XLISP-STAT now runs on a variety of platforms (including Macintoshes, PCs and selected Unix workstations), using native interface-management tools for each platform. A set of tools is provided for interface customization, including menu-, plot- and dialog-construction methods. XLISP-STAT is available free of charge.

QUAIL[7] (Quantitative Analysis In Lisp) is a quantitative programming environment in Common Lisp developed under R. W. Oldford at the Statistics Computing Laboratory, University of Waterloo. QUAIL is a sophisticated mathematical and statistics package providing a number of useful features including: extended arithmetic functions, a variety of array manipulation facilities, many specialized mathematical functions, probability calculations for an assortment of distributions, and statistical response models. QUAIL is CLOS-based and provides extensions to Common Lisp to support quantitative analysis. One distinctive feature of QUAIL is an easily-configurable user interface. QUAIL runs on Unix workstations and the Macintosh. Users must pay a minimal license fee.

CLASP has two important features that set it apart. The first is that its interface is easy for novices to use, especially students. The Unix version uses the same object-oriented, menu-based CLIM interface across all platforms and Lisp implementations: the Macintosh version of CLASP, while including most of

[7]Information about QUAIL may be obtained at `http://setosa.uwaterloo.ca/~ftp/Quail/Quail.html`.

the features of its Unix counterpart, has been customized for teaching a graduate-level course in empirical research methods. A second feature is that CLASP is designed specifically for the analysis of AI programs, providing support for the add-on experiment modules described above as well as an open architecture that is fully Common Lisp compatible, making it a substrate for portable extensions.

## 8 Current Status

CLASP has been ported to a variety of platforms, from its beginnings on the Texas Instruments Explorer™ Lisp machines to the latest native Macintosh interface and every other major Common Lisp implementation. Development of CLIP/CLASP continues, and is largely driven by user demand. We will continue to add useful statistical tests and data manipulation functions.

CLASP for Unix™ platforms uses CLIM 2.0 and is available by anonymous FTP from **`ftp.cs.umass.edu /pub/eksl/clasp`**. CLIP for all platforms is similarly available in **`pub/eksl/clip`**. A demonstration version of CLASP for the Macintosh can be found in **`pub/eksl/clasp`**. All versions include a manual and current release notes. The Macintosh version includes a brief tutorial. A tutorial featuring the Unix version can be found in **`pub/eksl/clasp-tutorial`**.

Comments, bugs and requests for more information can be sent to **`clasp-support@cs.umass.edu`**. Additional information on CLASP and CLIP is available on the World Wide Web at **`http://eksl-www.cs.umass.edu/clasp.html`**.

## 9 Conclusion

The purpose of this article is to demonstrate how CLIP and CLASP can help in doing statistical studies of the behavior of complex programs, such as Artificial Intelligence programs, using an example grounded in transportation planning. CLIP works directly with a user's simulator, helping the experimenter define the dependent measures, control the independent variables and run the experiment. CLASP is a statistics package and as such competes with many good statistics packages on the market. Its main advantage is that it is implemented in Common Lisp and CLIM, so that it can easily be combined with your simulator and with CLIP, allowing for a completely integrated experimental environment. It also allows the data file to be annotated with the names of variables, which can make data analysis more convenient. Finally, the article describes our more recent work on data analysis and experiment design built on top of the CLIP and CLASP ideas. We believe that such support for empirical science will be of significant benefit to the AI community.

### Acknowledgements

This research is supported by ARPA/Rome Laboratory under contracts #F30602-91-C-0076 and #F30602-93-C-0100. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. We thank Tim Oates for the use of his code and data, and Rob St. Amant for help in preparing this paper.

David Fisher wrote the first implementation of CLASP on the TI Explorer, using database software by Paul Silvey. Adam Carlson helped design and develop CLASP's first CLIM interface and did a great deal of work on the internal representations. Matthew D. Schmill, with help from Matt Cornell, developed the CLASP interface for the Macintosh. Matt Schmill also implemented the bootstrapping code. We thank them for their creativity and industry.

## References

[1] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Trial by fire: Understanding the design requirements for agents in complex environments. *AI Magazine*, 10(3):32–48, Fall 1989.

[2] Scott D. Anderson, Adam Carlson, David L. Westbrook, David M. Hart, and Paul R. Cohen. CLASP/CLIP: Common Lisp Analytical Statistics Package/Common Lisp Instrumentation Package. Technical Report 93-55, University of Massachusetts at Amherst, Computer Science Department, 1993.

[3] Tim Oates and Paul R. Cohen. Mixed-initiative schedule maintenance: A first step toward plan steering. In Mark H. Burstein, editor, ***ARPA/Rome Laboratory Knowledge-based Planning and Scheduling Initiative Workshop Proceedings***. Advanced Research Projects Agency and Rome Laboratory, February 1994. Also available as Technical Report 94-31, University of Massachusetts Computer Science Department.

[4] Tim Oates and Paul R. Cohen. Toward a plan steering agent: Experiments with schedule maintenance. In ***Proceedings of the Second International Conference on Artificial Intelligence Planning Systems***, 1994. Also available as Technical Report 94-02, University of Massachusetts Computer Science Department.

[5] Robert St. Amant and Paul R. Cohen. Preliminary system design for an EDA assistant. In ***Preliminary Papers of the Fifth International Workshop on AI and Statistics***, pages 502–512, 1995.

[6] John W. Tukey. ***Exploratory Data Analysis***. Addison-Wesley, 1977.

[7] Bradley Efron and Gail Gong. A leisurely look at the bootstrap, the jackknife, and cross-validation. ***The American Statistician***, 37(1):36–48, February 1983.

[8] Bradley Efron and Robert Tibshirani. Statistical data analysis in the computer age. ***Science***, 253:390–395, July 1991.

[9] Adele E. Howe and Paul R. Cohen. Understanding planner behavior. ***Artificial Intelligence***. To appear.

[10] Adele E. Howe. Finding dependencies in event streams using local search. In ***Preliminary Papers of the Fifth International Workshop on AI and Statistics***, pages 271–277, 1995.

[11] Tim Oates, Dawn E. Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In ***Preliminary Papers of the Fifth International Workshop on AI and Statistics***, pages 417–423, 1995.

[12] Paul R. Cohen, Adam Carlson, L. A. Ballesteros, and Robert St. Amant. Automating path analysis for building causal models from data. In ***Proceedings of the Tenth International Conference on Machine Learning***, pages 57–64. Morgan Kaufmann, 1993.

[13] H. B. Asher. ***Causal Modeling***. Sage Publications, 1983.

[14] C. C. Li. ***Path Analysis — A Primer***. Boxwood Press, 1975.

[15] Paul R. Cohen, Dawn E. Gregory, L. A. Ballesteros, and Robert St. Amant. Two algorithms for inducing structural equation models from data. In ***Preliminary Papers of the Fifth International Workshop on AI and Statistics***, pages 129–139, 1995.

[16] J. Pearl and T. Verma. A statistical semantics for causation. ***Statistics and Computing***, 2:91–95, 1991.

[17] P. Spirtes, C. Glymour, and R. Scheines. ***Causation, Prediction and Search***. Springer-Verlag, 1993.

[18] L. A. Ballesteros. Regression-based causal induction with latent variable models. In ***Proceedings of the Twelfth National Conference on Artificial Intelligence***, page 1426. American Association for Artificial Intelligence, AAAI Press/The MIT Press, 1994. Student Abstract.

[19] Paul R. Cohen. ***Empirical Methods in Artificial Intelligence***. MIT Press, 1995.

[20] Dawn E. Gregory and Paul R. Cohen. A function modeling approach to empirical science. To be presented at the 10th International Conference on Mathematical and Computer Modelling.

[21] Bolt Beranek and Newman, Inc. and ISX Corporation. Common prototyping environment testbed release 1.0: User's guide. BBN Systems and Technologies, 10 Moulton Street, Cambridge, MA 02138, 1993.

[22] Luke Tierney. XLISPSTAT. School of Statistics Report #528, University of Minnesota, 1988.

[23] Luke Tierney. ***LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics***. John Wiley & Sons, 1990.