In *Proceedings of the Sixth International IEEE Conference on Tools with Artificial Intelligence (1994)*, pp. 615-623.

Tools for Experiments in Planning

**Scott D. Anderson, Adam Carlson,
David Westbrook, David M. Hart
and Paul R. Cohen**

Computer Science Technical Report 95-01

Experimental Knowledge Systems Laboratory
Computer Science Department, Box 34610
University of Massachusetts
Amherst, MA 01003-4610

Replaces Technical Report 94-28

Abstract

The paper describes two separate but synergistic tools for running experiments on large Lisp systems such as Artificial Intelligence planning systems, by which we mean systems that produce plans and execute them in some kind of simulator. 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.

This research is supported by ARPA/Rome Laboratory under contract #'s F30602-91-C-0076 and F30602-93-C-0100.

1 Introduction

As planning problems become more complex, involving hundreds of objects and thousands of resources (e.g., ships, planes, tanks, satellites), researchers will need to turn to simulators, controlled experiments, and statistics to study the behavior of their systems. We will briefly describe one such simulator, called TransSim, and a controlled experiment that the Experimental Knowledge Systems Laboratory (EKSL) ran using it, but our real purpose in this description is to introduce 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 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 doing statistical operations that we have not anticipated.

This paper describes TransSim, Clip and Clasp, and presents an example of their use in an experiment, from initial description to final analysis.

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 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. 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.

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 is 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.

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 arrive on the day in question. If that probability exceeds some threshold, the demon assumes that the ship arrives. The demon also predicts how many ships will leave the port, 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 or ignoring 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 variance 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. Most of CLIP's functionality is directed towards extracting different kinds of information from your system—information that is calculated afterwards, collected periodically during execution, or possibly collected whenever some 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 [1], where everything is rigorously explained.)

To use CLIP to run an experiment, CLIP first needs to know how to run your simulator. 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. (This requirement may be lifted in future versions of CLIP, but the impact is minor. Most multi-threaded Lisps provide a `process-wait` function, which can be used to make the simulator seem like a single piece of code.) Between trials, CLIP will 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`.

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 with the 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. The `define-experiment` macro also provides ways for the user to run code of their own choosing during the experiment, at four distinct times:

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.

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. 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 on the screen saver.

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.

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 very much like **`defun`**, except that information added before the body is read by CLIP. The central issue in defining a clip is the time that it is run. (The code that is run is written in the **`defclip`** body and is entirely up to the user.) 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, which only makes sense for simulators that have a clock of some sort; CLIP will schedule the clip using the **`schedule-function`** specified in the **`define-simulator`** form. Other clips may need to run when some event occurs; this is accomplished by tying the clip to a function in your simulator, using a mechanism like the "advise" facility found in many Lisp implementations. The **`defclip`** form has syntax for tying the clip to a 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).

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, 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.

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.

One trouble with time series data is that it 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.

As with simple end-of-trial clips, clips for time series can be scheduled to run periodically, as with our once-a-day collection of information on queues, or can be triggered by events, as with our collection of information whenever a ship is loaded. The syntax of `defclip` makes all this relatively easy.

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:

Define the Simulator You tell CLIP how to run your system via the `define-simulator` macro.

Define the Clips You define a bunch of functions to report the data you want to collect. Most clips will simply return one value, which CLIP will write to the main data file, but you can also define clips that return multiple values, map over multiple objects, run multiple times, or any combination of these features.

Define the Experiment You finally put all of the pieces together by specifying the simulator to run, the experimental conditions (particularly the independent variables), special initialization/parameterization code, and the list of data values to be collected. A call to `write-current-experiment-data` is put here.

Run the Experiment Very little else needs to be specified when the experiment is finally run; usually only the output file, possibly the number of repetitions, and maybe one or two arguments that are referred to in the user's experiment code (`:before-experiment` and the other clauses).

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 [1].

An alternative to CLIP is the METERS system, developed by Bolt, Beranek and Newman, Inc., for use in the ARPA/RL Planning Initiative's Common Prototyping Environment (CPE) [2]. METERS is particularly useful for collecting and filtering time-series data from distributed systems.

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), graphing software (now replaced by SCIGRAPH, by Bolt, Beranek and Newman, Inc.). We have a convenient graphical user interface implemented in CLIM, and a programmatic interface so that the CLASP functions can be called by the user if the 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 1, comprises four areas: the menus, the datasets, the results, and the notebook:

Menus The CLASP menus will appear across the top of the window. The menus, which will be discussed below, are: File, Graph, Describe, Test, Manipulate, Transform 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 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 this pane become mouse-sensitive under those circumstances. 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 selecte 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, that object goes into a menu of results. 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. 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.

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 one 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 menu, as given below. The description just summarizes the kinds of commands; 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 (such as graphs or tables) to PostScript files. (Currently, CLASP uses CLIM 1.1, which does not produce encapsulated PostScript (EPS). We will soon complete an implementation using CLIM 2.0, which will produce EPS, making it easy to insert graphs into other documents.)

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.

Test This menu contains the inferential statistics that were omitted from the previous 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. One other, called the *d*-test may be unfamiliar, since it is a bootstrap statistic to compete with the *t*-test. Bootstrap statistics [3, 4] replace the parametric and distributional assumptions of statistics like the *t*-test with an empirical approach using computerized resampling of the data. The *d*-test is used just like the *t*-test, especially when the data don't satisfy the normality and equal-variance assumptions of the *t*-test. In the near future, we hope to implement bootstrap variants on most common statistical functions.

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 categorial variable from that data. A categorial 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 categorial 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 executed 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**.

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 important, 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 of using 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 [5, 6, 7]. 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.

```
(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 at the top of the next page. 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 **:ivs** 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 the values of the independent variables to the appropriate global variables of the 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)
  :ivs               ((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 property-list style. 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*)))))
```

The experiment is run by executing the following code. The **:repetitions** clause says how many trials to run under each condition (combination of levels of the independent variables).

```
(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 1 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 ($F = 1.6, p = 0.2$).

6 Current status

CLIP and CLASP may be obtained by anonymous ftp from ftp.cs.umass.edu. CLIP can be found under the directory pub/eksl/clip; CLASP is under pub/eksl/clasp. Manuals are included in both of these directories. A tutorial on CLASP is available under pub/eksl/clasp-tutorial.

Development of CLIP/CLASP continues, and is largely driven by user demand. We will continue to add useful statistical tests and data manipulation functions. Known limitations include problems with encapsulated PostScript output from CLASP due to CLIM 1.1 and cosmetic glitches in display and input editing, which are due in part to CLIM 1.1. CLIP could use a graphical user interface for defining experiments. Comments, bugs and new feature requests can be sent to clasp-support@cs.umass.edu. For more information about CLIP/CLASP, contact David Hart (dhart@cs.umass.edu, 413-545-3278).

7 Conclusion

The purpose of this paper is to demonstrate how CLIP and CLASP can help in doing experimental studies in Artificial Intelligence generally, 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 advantages are 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. We believe that such support for empirical science will be of significant benefit to the AI community.

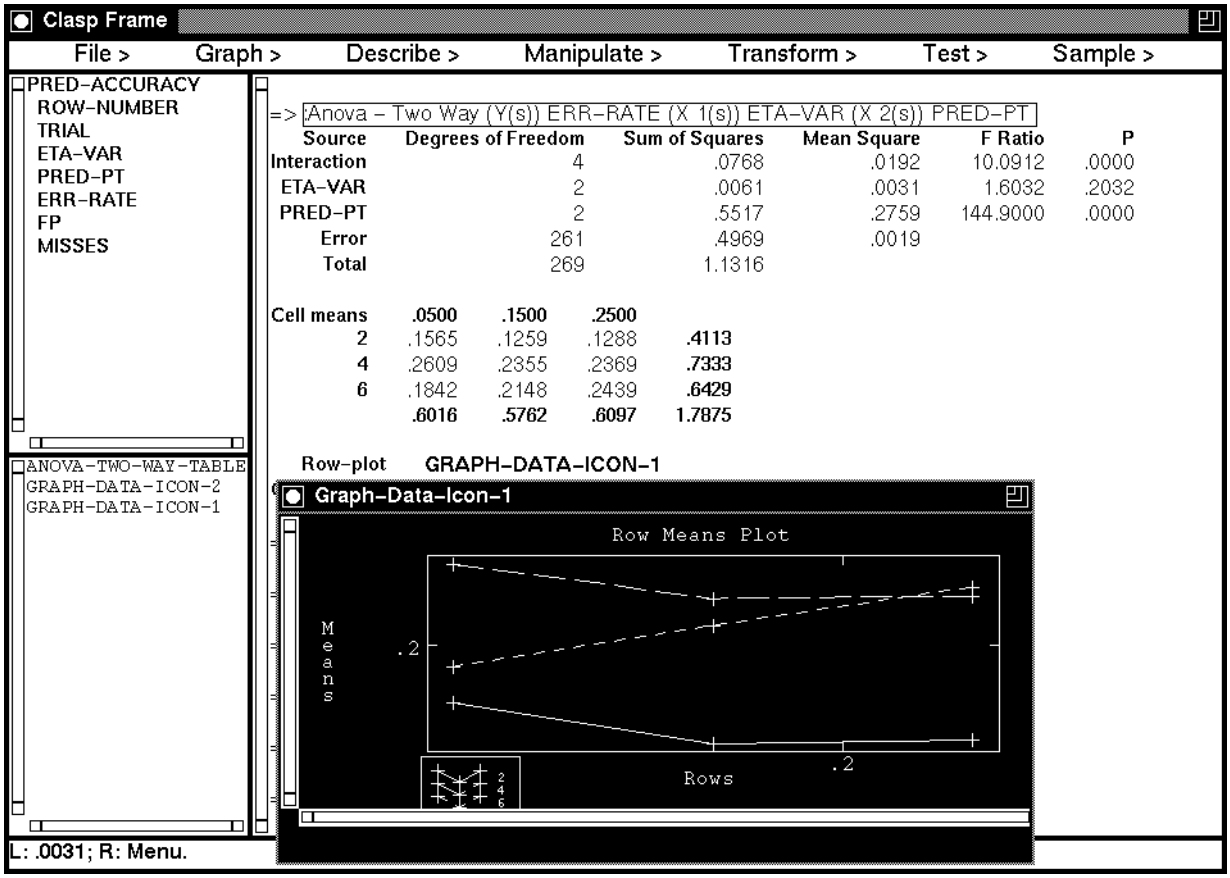

Figure 1: Excerpt from sample interaction with CLASP

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 a helpful reading of the paper.

References

[1] 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.

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

[3] 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.

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

[5] Tim Oates and Paul R. Cohen. Humans plus agents maintain schedules better than either alone. Technical Report 94-03, University of Massachusetts at Amherst, Computer Science Department, 1994.

[6] 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.

[7] 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.